Contraintes Globales :
Aspects Algorithmiques et Déclaratifs

Rapport d'Habilitation à Diriger des Recherches

Nicolas Beldiceanu

28 avril 2003

Université de Paris VI
4, place Jussieu
75252 Paris cedex 05, France

Pierre Cointe	:	examinateur
Alain Colmerauer	:	rapporteur
Max Fontet	:	président
Claude Le Pape	:	examinateur
Claude Girault	:	examinateur
Michel Minoux	:	rapporteur
Pascal Van Hentenryck	:	rapporteur

à Marie-Hélène,
Marie-Claire,
Emmanuelle,
Florence,

et Mats.

Remerciements

Mes remerciements vont plus particulièrement à :

- *Abderrahmane Aggoun* qui m'a introduit au système CHIP et pour notre longue collaboration,
- *Jean Berstel* pour ses cours d'initiation aux problèmes discrets (graphes, automates, mots),
- *Eric Bourreau* pour son esprit critique,
- *Mats Carlsson* pour m'avoir accueilli à SICS et pour m'avoir encouragé à classer les contraintes globales,
- *Philippe Charlier* pour nos interactions qui ont motivé plusieurs contraintes,
- *Pierre Cointe*, Professeur à l'école des Mines de Nantes, pour avoir accepté d'être membre de ce jury,
- *Alain Colmerauer*, Professeur à l'université de Méditerranée, pour avoir accepté d'être rapporteur de ce mémoire,
- *Evelyne Contejean* pour un travail en commun pendant cinq ans,
- *Max Fontet*, Professeur à l'université Pierre et Marie Curie, pour avoir accepté d'être président du jury,
- *Hervé Gallaire* pour son soutien lorsque j'avais fini ma thèse,
- *Claude Girault*, Professeur à l'université Pierre et Marie Curie, pour son soutien amical et pour ses conseils au sujet de ce mémoire,
- *Jean-Louis Laurière* pour son travail sur ALICE qui m'a communiqué l'envie d'entrer dans le domaine des contraintes,
- *Claude Le Pape*, Directeur recherche et développement chez ILOG, pour avoir accepté d'être membre de ce jury et pour ses commentaires sur ce mémoire,
- *Per Mildner* pour sa relecture attentive de mes articles,
- *Michel Minoux*, Professeur à l'université Pierre et Marie Curie, pour avoir accepté d'être rapporteur de ce mémoire,
- *Jacques Pitrat* pour ses cours et ses idées dans le domaine du méta,
- *Emmanuel Poder* pour son travail concernant la contrainte *cumulative-trapèze*,
- *Qi Guo* et *Maud Stavenor* pour leur soutien,
- *Helmut Simonis* pour avoir été l'un des tous premiers à montrer comment vraiment utiliser les contraintes dans des applications industrielles,
- *Sven Thiel* pour ses réponses toujours promptes à mes questions,
- *Pascal Van Hentenryck*, Professeur à l'université de Brown, pour avoir accepté d'être rapporteur.

Finalement, je remercie toutes les personnes avec qui j'ai eu des échanges au sujet des contraintes globales, en particulier Magnus Ågren, Ernst Althaus, Gregor Baues, Pascal Brisset, Peter Chan, Frédéric Deces, Mehmet Dincbas, François Fage, Pierre Flener, Yan Georget, David Hanak, Narendra Jussien, Waldemar Kocjan, Per Kreuger, Krzysztof Kuchcinski, Michel Leconte, Michael Marte, Nicolas Museux, Justin Pearson, Guillaume Rochart, Xavier Savalle, Péter Szeredi.

Sommaire

Note

La bibliographie située à la fin de ce mémoire permet de retrouver l'ensemble de mes articles. Pour des raisons de confidentialité vis-à-vis de COSYTEC*, je n'ai utilisé dans ce mémoire, pour la partie concernant* CHIP*, que des informations provenant d'articles publiés* [AGGOUN & BELDICEANU 93b], [BELDICEANU & CONTEJEAN 94] *accessibles à tout un chacun. Finalement, je précise ci-dessous les parties des articles provenant pour une grande part du travail d'un coauteur différent de moi-même :*

– *dans* [AGGOUN & BELDICEANU 90] *l'utilisation du concept de « time-stamp» dans le cadre de la gestion mémoire liée aux contraintes a été inventée par Abderrahmane Aggoun,*
– *dans* [AGGOUN & BELDICEANU 93a] *l'écriture du compilateur* CHIP *a été réalisée par Abderrahmane Aggoun,*
– *dans* [BELDICEANU, CARLSSON & THIEL 02] *les parties concernant le calcul de la borne supérieure et du regret associé proviennent de Sven Thiel,*
– *la deuxième partie de l'article* [BELDICEANU, CARLSSON & THIEL 03] *concernant le alldifferent provient de Sven Thiel,*
– *dans* [BELDICEANU, GUO & THIEL 01] *l'algorithme de filtrage*[1] *traitant le non-recoupement de polygones vient de Sven Thiel,*
– *dans* [PODER, BELDICEANU & SANLAVILLE 02] *l'invention de l'algorithme calculant la partie obligatoire a été le fruit du travail d'Emmanuel Poder.*

[1] Un algorithme de filtrage est un algorithme localisant des valeurs ne pouvant pas être prises par une variable en fonction du fait qu'une contrainte mentionnant la variable en question doit être satisfaite.

1 Introduction

Initialement introduites dans les années 60 aux États-Unis par la communauté intelligence artificielle les contraintes se sont lentement établies comme une discipline à part entière. L'idée clef de cette discipline consiste à remplacer l'écriture de programmes procéduraux par la description explicite de ce que l'on veut obtenir : on précise les contraintes que l'on veut vérifier et laisse au système le soin de calculer, à partir de ces informations partielles, une solution satisfaisant les contraintes.

Les premiers systèmes de programmation par contraintes imposèrent des restrictions fortes sur la nature des contraintes que l'on pouvait à priori exprimer. La confrontation au monde industriel amena l'introduction de nouvelles contraintes plus proches des problématiques rencontrées. Ce fut l'occasion de remplacer des procédures de résolutions générales par des algorithmes adaptés aux différents types de contraintes traitées. Ces contraintes furent dénommées *contraintes globales* et constituent le domaine de mes recherches depuis ma thèse.

Sans pour autant quitter complètement le domaine de l'intelligence artificielle, ce passage du général au particulier s'est accompagné d'un glissement progressif vers la recherche opérationnelle. Cela a entraîné une tension entre les aspects déclaratifs et les aspects procéduraux qui s'est autrefois manifestée par une incompréhension mutuelle entre les deux communautés.

A l'heure actuelle la programmation par contraintes est souvent mal comprise comme une simple juxtaposition ou une réappropriation de certaines techniques venant de différentes disciplines. On assiste couramment à une explosion du nombre de contraintes globales et des domaines de calculs associés (e.g. entiers, rationnels, ensembles, …) qui rend extrêmement difficile une gestion harmonieuse des différents aspects liés aux contraintes (algorithmes de filtrage, visualisation, heuristiques spécialisées, coupes pour la programmation linéaire, …).

Face au constat précédent, l'objectif premier de la programmation par contraintes est, à mes yeux, d'essayer de tisser des liens entre les aspects déclaratifs et les aspects procéduraux. Pour cette raison, après avoir initialement uniquement travaillé sur les aspects algorithmiques concernant le traitement efficaces de certaines contraintes, je me suis par la suite également consacré aux aspects déclaratifs consistant à rendre explicites des informations qui, jusqu'à présent, étaient enfouies au cœur des algorithmes et des contraintes. Le but ultime étant de remplacer une jungle de contraintes ad hoc par une vision unifiée dans laquelle on établit clairement les liens

permettant de passer des aspects déclaratifs aux aspects procéduraux. Cette vision passe, d'une part par la classification des contraintes et leur description en termes de constituants élémentaires, et d'autre part par la recherche systématique des liens entre les propriétés de ces constituants élémentaires et les algorithmes de résolution associés aux contraintes.

Pour aborder les questions précédentes, deux domaines sont importants : d'une part les mathématiques discrètes et l'algorithmique associée (dénombrement [TUCKER 80], graphes [BERGE 70] [BERGE 87], [GONDRAN & MINOUX 85] mots [CROCHEMORE, HANCART, LECROQ 01], géométrie [PREPARATA & SHAMOS 85], [DE BERG, VAN KREVELD, OVERMARS, SCHWARZKOPF 97], [MEHLHORN & NÄHER 99]), et d'autre part la partie de l'intelligence artificielle concernant la déclarativité [PITRAT 93] et la réflexivité. Cette partie consiste à expliciter les informations afin de pouvoir écrire des programmes plus flexibles pouvant s'appliquer dans un plus grand nombre de cas.

Ce mémoire est organisé en trois parties. La première partie contient un état de l'art dans le domaine des contraintes globales. Après un rappel historique des faits marquants l'évolution de ce domaine je passe en revue les systèmes contenant des contraintes globales et leur domaines d'applications respectifs. Finalement, je termine par une bibliographie thématique sur les contraintes globales. Cette dernière présente, pour chaque thème où interviennent les contraintes globales, les idées principales et les articles correspondant. En dehors des contraintes globales proprement dites, ces thèmes concernent les domaines suivants :

- les contraintes globales avec fonction de coût,
- les contraintes globales et la programmation linéaire,
- la relaxation de contraintes globales,
- le traitement de certaines conjonctions de contraintes globales,
- les contraintes globales et les méta-heuristiques.

Dans la deuxième partie de ce mémoire je présente mes principales contributions dans le domaine des contraintes globales en me focalisant sur les aspects algorithmiques. Dans un premier temps j'explique le contexte m'ayant conduit à introduire le concept de contrainte globale. Puis je développe les quatre thèmes suivants :

- **les contraintes globales dans le domaine de l'ordonnancement avec contraintes de capacité** [AGGOUN & BELDICEANU 92], [AGGOUN & BELDICEANU 93b], [SIMONIS & BELDICEANU 99], [BELDICEANU & CARLSSON 02], [PODER, BELDICEANU & SANLAVILLE 02]**.** C'est le domaine dans lequel la programmation par contraintes a eu le plus de succès. En effet, bien que l'importance économique de ces problèmes ait suscité depuis bientôt trente ans une recherche abondante, c'est la programmation par contraintes qui a permis de fournir à l'industrie des

composants réutilisables par des ingénieurs ne connaissant pas la recherche opérationnelle. La difficulté principale des problèmes d'ordonnancement provenant du fait que les tâches doivent partager des ressources, j'ai introduit des contraintes prenant en compte cet aspect. Une des originalités dans ce contexte concerne le fait que j'ai également traité le cas important ou la consommation de ressources par les tâches est variable dans le temps.

– **les contraintes globales et la géométrie** [BELDICEANU & CONTEJEAN 94], [BELDICEANU 00a], [BELDICEANU & CARLSSON 01b], [BELDICEANU, GUO & THIEL 01], [BELDICEANU, CARLSSON & THIEL 03], [BELDICEANU & CARLSSON 01c]**.** Il s'agit d'un domaine important vis-à-vis de la programmation par contraintes. En effet, l'introduction de contraintes géométriques permet d'étendre le champ applicatif de la programmation par contraintes. De plus, l'utilisation de techniques provenant de la géométrie algorithmique s'avère judicieuse pour le traitement de certaines contraintes : une contrainte pouvant souvent être vue comme un ensemble d'objets géométriques sur lesquels il est possible de raisonner globalement.
– **techniques spécifiques pour les algorithmes de filtrage** [BELDICEANU & CARLSSON 01a], [BELDICEANU & CARLSSON 01c], [CARLSSON & BELDICEANU 02a], [CARLSSON & BELDICEANU 02b]**.** J'ai regroupé dans cette rubrique des techniques originales basées sur le comptage d'événements particuliers et sur la construction d'automates reconnaissant le langage associé aux solutions acceptées par une contrainte.
– **contraintes avec fonction de coût** [BELDICEANU, CARLSSON & THIEL 02]**.** La programmation par contraintes a souvent été dénoncée pour sa faiblesse dans le domaine de l'optimisation. Pour cette raison, plusieurs chercheurs tels que Philippe Baptiste, Yves Caseau, Torsten Fahle, Claude Le Pape, François Laburthe ou Michela Milano ont introduit des contraintes «d'optimisation». Dans ce domaine je me suis intéressé à une fonction de coût, qui à notre connaissance, n'a pas été traitée en recherche opérationnelle, mais qui a cependant de nombreuses applications pratiques. Il s'agit en l'occurrence de la *somme des poids associés aux valeurs distinctes* prises par un ensemble de variables de décision.

Dans la troisième partie de ce mémoire je présente ma contribution dans le domaine des contraintes globales en se focalisant sur les aspects déclaratifs [BELDICEANU 00b], [BELDICEANU 00c], [SIMONIS, AGGOUN, BELDICEANU & BOURREAU 00]. Je présente mon travail dans le domaine de la classification des contraintes globales, dans le domaine de la communication entre contraintes globales et finalement dans le domaine de la mise au point de programmes dans le cadre de la programmation par contraintes.

Je présente à chaque fois ces articles en les replaçant dans le contexte ayant conduit à leur création. Avant de passer au chapitre 2, j'introduis le concept de *contraintes globales*.

Introduction du concept de contrainte globale

Bien qu'il n'existe pas de définition formelle de la notion de contrainte globale, une *contrainte globale* est souvent assimilée à une condition possédant une structure liant plus de deux variables domaine. Une *variable domaine* est une variable prenant sa valeur parmi un ensemble fini d'entiers. Un *algorithme de filtrage* associé à une contrainte globale est un algorithme déterminant les valeurs impossibles à prendre pour les variables mentionnées par la contrainte. Il s'agit des valeurs qui, si elles étaient prises, conduiraient immanquablement au fait que la contrainte ne puisse pas être satisfaite. Un algorithme de filtrage est dit *complet* s'il peut identifier toutes ces valeurs impossibles.

Illustrons les notions précédentes à l'aide de la contrainte *alldifferent*. Cette contrainte impose que les variables d'un ensemble donné prennent des valeurs entières distinctes. La structure associée à la contrainte *alldifferent* correspond à une clique de contraintes de diségalités (i.e. toutes les variables sont deux à deux différentes).

Considérons maintenant quatre variables V_1,V_2,V_3 et V_4 prenant respectivement leur valeur dans les ensembles {2,3}, {2,3}, {1,3} et {1,2,3,4} sur lesquelles on impose la contrainte *alldifferent*. L'algorithme complet de filtrage décrit par Jean Charles Régin [RÉGIN 94] détermine directement l'ensemble des valeurs impossibles pour les variables. Cela conduit à fixer V_3 à 1 et V_4 à 4. En revanche, si l'on traite chacune des contraintes de diségalité de manière indépendante, on perd la possibilité d'identifier directement l'ensemble des valeurs impossibles.

Terminons cette introduction en mentionnant que la notion de globalité peut, de manière informelle, s'interpréter aux niveaux suivants :

- Au niveau *sémantique* cela signifie qu'une contrainte ne peut s'exprimer comme la conjonction d'un nombre de contraintes plus élémentaires[2]. Ce n'est pas de cas de la contrainte *alldifferent* que nous venons de voir. Elle peut en effet se voir comme la conjonction d'un ensemble de contraintes de diségalités entre deux variables. En revanche la contrainte *period*[3] ne peut pas se décomposer en une conjonction de contraintes élémentaires. Cette difficulté au niveau de l'expression d'une contrainte se rencontre malheureusement bien souvent dans la pratique.
- Au niveau *déductif* cela indique que, bien qu'une contrainte C puisse s'exprimer comme une conjonction de contraintes plus élémentaires, on a besoin de rester au niveau de C si l'on veut élaguer le plus possible le

---

2 On suppose que l'on a pas le droit d'introduire d'autres variables.

3 La contrainte period(P,[V_0,…,V_{m-1}]) dans laquelle P est une variable domaine et V_0,…,V_{m-1} une séquence de variables domaine est satisfaite si P est la période de la séquence V_0,…,V_{m-1}. La période d'une séquence V_0,…,V_{m-1} est le plus petit entier p tel que $V_i= V_{i+p}$ pour tout i dans 0,1,…,m–p–1.

domaine des variables intervenant dans C. C'est par exemple le cas de la contrainte *alldifferent*. En effet lorsque l'on exprime la contrainte *alldifferent* comme la conjonction d'un ensemble de contraintes de diségalités on perd irrémédiablement l'élagage fournit par l'algorithme de filtrage associé à la contrainte *alldifferent* [RÉGIN 94].

- Au niveau *algorithmique* cela signifie que, bien qu'une contrainte C puisse s'exprimer comme une conjonction de contraintes plus élémentaires, on a besoin de rester au niveau de C si l'on veut garder la complexité spatiale ou temporelle offerte par l'algorithme de filtrage associé à C.

2 Émergence des contraintes et premières contributions

Ce chapitre commence par un bref aperçu historique du développement des contraintes en se focalisant sur les contraintes portant sur des variables entières. C'est en effet le domaine auquel sont rattachées les contraintes globales. Il s'agit d'un domaine particulièrement actif avec de nombreuses applications industrielles réussies dans les secteurs du transport, de l'ordonnancement, de la bio-informatique et des réseaux reposant sur l'existence de plusieurs fournisseurs de composants tels que ILOG (`http://www.ilog.com/`), COSYTEC (`http://www.cosytec.com/`), Koalog (`http://www.koalog.com/php/jcs.php`) ou SICS (`http://www.sics.se/`). Enfin notons que, pour des raisons d'indépendance, plusieurs industriels tels que Thales ou Bouygues développent leur propre système.

Dans la dernière partie de ce chapitre, je répertorie les principaux systèmes contenant des contraintes globales et donne une bibliographie thématique dans le domaine des contraintes globales. Pour chaque système j'indique les secteurs applicatifs lui correspondant. Chaque contrainte est introduite par une brève définition, suivie de références sur les algorithmes qu'elle utilise[4] ainsi que les applications où elle est typiquement employée.

Avant de débuter notre aperçu historique centré sur les contraintes globales, il est cependant bon de rappeler que, de par leur aspect pluridisciplinaire, les contraintes sont présentes dans les deux grandes branches de l'informatique que sont l'intelligence artificielle et la recherche opérationnelle.

En ce qui concerne l'intelligence artificielle, on retrouve les contraintes dans les domaines suivants :

- la programmation logique [COLMERAUER 79], [KOWALSKI 79] [GALLAIRE 87] et ses extensions aux contraintes [JAFFAR & LASSEZ 87], [VAN HENTENRYCK 89], [JAFFAR, MICHAYLOW, STUCKEY & YAP 92],
- la logique et la réécriture [NIEHREN, TREINEN & TISON 00],
- la méta-programmation [PITRAT 93, pages 163–187],
- les réseaux de contraintes [DECHTER & PEARL 87].

D'un autre côté, pour la recherche opérationnelle [GENDREAU 99], on retrouve les contraintes dans les domaines suivants :

- l'ordonnancement [BAPTISTE, LE PAPE & NUIJTEN 01],
- la programmation linéaire [BOCKMAYR & KASPER 98],
- les langages de modélisation de problèmes [VAN HENTENRYCK 99],

4 Lorsque ces algorithmes ont été publiés.

– les méta-heuristiques [NAREYEK 01], [MICHEL & VAN HENTENRYCK 02],
– les algorithmes hybrides [CASEAU & LABURTHE 96a].

Nous reviendrons par la suite sur les liaisons de chacun de ces domaines avec les contraintes globales.

Finalement, mentionnons en dehors des branches de l'intelligence artificielle et de la recherche opérationnelles les deux domaines suivants :

– les bases de données [GAEDE, BRODSKY, GÜNTHER, SRIVASTAVA, VIANU & WALLACE 97],
– les langages concurrents [SARASWAT 87] et [HARIDI, VAN ROY, BRAND & SCHULTE 98].

Nous retraçons maintenant l'évolution du concept de contraintes en indiquant les étapes marquantes que furent l'introduction de Prolog [COLMERAUER & ROUSSEL 93], d'ALICE [LAURIÈRE 76] et de CHIP [DINCBAS, VAN HENTENRYCK, SIMONIS, AGGOUN, GRAF & BERTHIER 88a]. Puis nous étudions la diffusion des contraintes dans l'industrie en montrant comment cela nous amena à introduire le concept de contrainte globale.

2.1 Émergence, évolutions et applications du concept de contraintes

C'est au début des années 60 que sont apparues les contraintes. A l'époque le système Sketchpad [SUTHERLAND 63] permettait de construire de manière interactive une figure géométrique en imposant des contraintes sur le placement respectif de différents points de cette figure. Un successeur de Sketchpad fut le système ThingLab de Alan Borning [BORNING 77] qui permettait de compiler un ensemble de contraintes. Notons également le développement au MIT du langage CONSTRAINTS [STEELE & SUSSMAN 80] utilisé pour des applications dans l'analyse de circuits.

L'introduction du non-déterminisme et des variables logiques dans les langages : PROLOG

La naissance de Prolog [COLMERAUER & ROUSSEL 93] remonte au tout début des années 70 dans le contexte de l'élaboration d'outils d'analyse du langage naturel. Du point de vue des langages de programmation, les originalités principales de Prolog sont, d'une part l'introduction du non-déterminisme, et d'autre part l'introduction de variable logiques (i.e. des inconnues) sur lesquelles le programmeur pouvait imposer des relations. Il s'agissait à l'époque d'un saut conceptuel important, car les variables des langages de programmation traditionnels ne désignaient que des adresses mémoire. Ces variables et les relations qui les relient peuvent également s'interpréter comme

des contraintes. L'introduction de la primitive *geler*[5] dans Prolog II [COLMERAUER 82] fut un point important car elle permettait d'introduire la notion de coroutine. Les chercheurs du domaine de la programmation logique par contraintes se sont bien vite aperçus que cela facilitait l'introduction des contraintes dans Prolog. Par la suite, différents domaines de calculs tels que les Booléens, les nombres rationnels et les chaînes de caractères sur lesquelles on pouvait poser des contraintes ont mené à Prolog III [COLMERAUER 90] et son successeur Prolog IV [BENHAMOU et al. 96].

Un précurseur dans le domaine discret : ALICE

Dans le cadre des contraintes portant sur les variables entières, un des précurseurs à plus d'un titre est le système ALICE [LAURIÈRE 76] qui date de bientôt trente ans. Parmi les originalités de ce système on peut citer :

– un langage de modélisation de problèmes ; Pendant les vingt-cinq années qui suivirent, la communauté contrainte a ignoré ce problème. En effet les contraintes étaient directement intégrées dans des langages de programmation tels que Prolog, Lisp, C, C++ ou Java. Ce n'est que très récemment que dans un souci de rendre les contraintes accessibles à un plus large public ignorant la programmation que l'on est revenu aux langages de modélisation dans le domaine des contraintes. Ce fut alors l'avènement du langage OPL [VAN HENTENRYCK 99] sur lequel nous reviendrons.
– l'utilisation de la notion de relation contrainte entre deux ensembles ; Il s'agit d'un outil puissant pour modéliser de manière naturelle certains problèmes dans lesquels on demande de chercher une relation vérifiant des propriétés telles que l'injection, la surjection ou la bijection. Cette idée a récemment été reprise dans la thèse de Brahim Hnich [HNICH 03].
– en dehors des contraintes arithmétiques usuelles la contrainte *alldifferent* ainsi que certaines contraintes sur les graphes ; elles permettent de modéliser de façon concise (sans passer par des variables 0/1) un certain nombre de contraintes intervenant de manière récurrente dans un grand nombre de problèmes discrets. Elles peuvent être considérées comme les premières contraintes globales.
– un traitement global de certaines conjonctions de contraintes comme par exemple la conjonction d'une contrainte *alldifferent* et d'une contrainte d'égalité entre un terme linéaire et une variable domaine ; cela permettait d'évaluer la valeur minimale ou maximale d'une variable présente dans un

[5] Elle permettait de reporter l'application de certaines règles jusqu'au moment où une variable donnée n'est plus libre.

terme linéaire en évitant de positionner les autres variables à une même valeur (minimale ou maximale) et d'obtenir ainsi des bornes plus fines[6].

– un choix dynamique de la méthode d'énumération en fonction du contexte ; En dehors de quelques articles limités à un contexte très restreint [EPSTEIN, FREUDER, WALLACE, MOROZOV & SAMUELS 02], ce choix audacieux n'a pas été repris dans les systèmes de programmation par contraintes actuels. On est en effet forcé d'utiliser une heuristique prédéfinie ou de programmer sa propre heuristique. En pratique cela restreint le public pouvant utiliser les contraintes aux personnes sachant programmer.

D'un autre côté les limitations principales d'ALICE sont à nos yeux :

– un jeu trop restreint de contraintes ; à l'époque ce fait est passé inaperçu car les problèmes abordés ne comportaient pas de contraintes opérationnelles complexes. Cela n'est malheureusement plus le cas de nos jours ou l'on veut modéliser de plus en plus précisément tous les aspects d'un problème. C'est le nombre trop important de contraintes potentielles qui a motivé notre recherche dans le méta.
– une volonté délibérée[7] d'écarter des algorithmes déjà connus à l'époque, tel que par exemple la recherche d'un couplage de cardinalité maximale dans un graphe biparti [HOPCROFT & KARP 73][8] ; Il s'agit pourtant d'un algorithme indispensable pour un traitement efficace de la contrainte *alldifferent*. Un cas similaire concerne le problème d'affectation et l'algorithme Hongrois. De notre côté nous n'écartons pas les algorithmes, mais vu le nombre trop important de contraintes, nous recherchons des algorithmes génériques applicables à des familles de contraintes.
– trop d'optimisations codées en dur dans le noyau de contraintes concernant le traitement de la conjonction de certaines contraintes particulières. Vu le grand nombre de conjonctions potentielles cela a conduit à un noyau de contraintes ingérable et cela a motivé sa reprise dans un cadre plus déclaratif [PITRAT 93, chapitre 12] sur lequel nous ne disposons malheureusement que de très peu d'informations. A l'heure actuelle le noyau type d'un système de contraintes est relativement petit car il ne gère que les modifications des domaines des variables et le réveil des contraintes. Cependant le problème important d'une gestion efficace générique d'une conjonction de contraintes reste peu étudié mis à part le

6 Par exemple, considérons la conjonction de contraintes $X_1+X_2=X_3 \wedge X_1 \neq X_2$ et supposons que le minimum de toutes les variables soit égal à 1. En tenant compte du fait que X_1 et X_2 prennent une valeur différente on déduit que le minimum de X_3 est égal à 3 (et non 2).

7 Le but d'ALICE étant de montrer qu'un système général pouvait résoudre aussi bien des problèmes que des algorithmes spécialisés, Jean-Louis Laurière avait volontairement écarté ce genre d'algorithmes. Ce fait est précisé en conclusion de sa thèse d'état [LAURIÈRE 76].

8 Bien qu'il existe d'autres algorithmes pour ce problème, nous citons ce dernier car il était connu en 1976.

mécanisme de communication entre contraintes sur lequel nous reviendrons plus tard [BELDICEANU 00b].

Les tous premiers débuts dans le monde industriel

Après l'avènement d'ALICE, il s'est écoulé environ un délai de huit années avant que le monde industriel ne commence vraiment à s'intéresser aux contraintes. En 1985, l'intérêt industriel au niveau des contraintes a suscité le développement du langage CHIP [DINCBAS, VAN HENTENRYCK, SIMONIS, AGGOUN, GRAF & BERTHIER 88a] qui a débuté dans les années 85 à l'European Computer Research Centre, un centre de recherche commun à Bull, Siemens et ICL. C'est le langage Prolog [COLMERAUER & ROUSSEL 93] qui a été sélectionné à l'époque comme support de CHIP étant donné sa popularité due à son choix par les Japonais comme langage de 5ième génération. Le mécanisme d'unification de Prolog a été étendu aux trois domaines de calcul suivants : les domaines finis, les booléens et les rationnels. Dans chacun des ces domaines, CHIP offrait des contraintes ainsi que les résolveurs correspondants. Ils ont commencé à être utilisés pour développer des prototypes dans les domaines de la production [DINCBAS, SIMONIS & VAN HENTENRYCK 88b], des découpes [DINCBAS, SIMONIS & VAN HENTENRYCK 88c], des problèmes financiers [BERTHIER 88] et de la validation de circuits [SIMONIS, NGUYEN & DINCBAS 88].

Dans le domaine qui nous intéresse plus particulièrement ici, à savoir les domaines finis, on peut noter l'apparition de la contrainte *element* [VAN HENTENRYCK & CARILLON 88] qui fut, en dehors de la contrainte *alldifferent*, l'une des premières contraintes globales. Nous reviendrons à la fin de cette section sur ces deux contraintes qui se retrouvent dans la majorité des applications pratiques et des systèmes.

Le premier travail plus systématique sur les contraintes globales remonte à mon arrivée à l'ECRC en 1988. J'ai commencé par me focaliser sur le développement de contraintes globales telles que *alldifferent* (sans la réécrire sous la forme de diségalités entre deux variables) [AGGOUN & BELDICEANU 93a] et sur une extension de la contrainte de *alldifferent* entre des paires de variables [BELDICEANU 90]. Cela a permis dès 1988 de résoudre le problème des *n*-reines pour des tailles allant jusqu'à *n*=1600 alors qu'à l'époque les systèmes de programmation par contraintes ne dépassaient pas les 100 reines[9]. En 1989 la contrainte de diségalité entre des paires de variables a permis de prouver l'absence de solution pour deux carrés latins orthogonaux d'ordre 6 (1 heure 20 sur un SUN 3/260), ce qui, là encore était hors de portée des systèmes de l'époque. Fin 1989, je me suis également

[9] En utilisant la modélisation avec trois contraintes *alldifferent* que j'avais trouvée à l'époque.

intéressé à un traitement global pour une contrainte *alldifferent* et un ensemble de contraintes d'inégalité[10]. Cette nouvelle contrainte combinant des diségalités et des inégalités a été utilisée avec succès dans des problèmes de routages [SIMONIS 90].

En 1989 la société BULL a repris la partie domaine finie de CHIP en lui rajoutant une interface avec le langage C pour la rendre plus attrayante auprès d'un public industriel. Cela a conduit au langage CHARME, le premier système commercial de programmation par contraintes sur les variables entières.

La diffusion des contraintes dans l'industrie

C'est au début des années 1990 que Jean-François Puget introduisit le résolveur de contraintes PECOS [PUGET 92] qui offrait un système de contraintes sur les domaines finis et sur les variables ensemblistes finies dans le cadre du langage LISP. Le choix de ce langage fut motivé à l'époque par le fait que c'était le langage utilisé au sein de la société ILOG. Par la suite, comprenant l'intérêt industriel d'une bibliothèque de contraintes, PECOS fut réécrit sous la forme d'une bibliothèque C++. La diffusion à une plus large échelle des contraintes dans l'industrie fut incontestablement le fait de cette bibliothèque C++ d'ILOG [PUGET 94] sur les domaines finis. Bien que, ne présentant initialement aucune originalité particulière au niveau de la résolution des contraintes dans les domaines finis, elle a pour la première fois montré comment intégrer les contraintes avec les objets. Elle a ainsi ouvert la possibilité de programmer dans un langage dans l'air du temps ce qui, à l'inverse des langages Lisp ou Prolog, rendait les contraintes attractives auprès d'un plus large public. Dans le domaine des problèmes discrets, on a cependant bien vite ressenti les limitations des systèmes de programmation par contraintes de l'époque. En effet tous ces systèmes étaient principalement basés sur des contraintes locales[11]. Cela conduisait parfois à des problèmes de performance tant du point de vue mémoire que du temps de calcul. Cet état de fait a motivé la continuation du travail que j'avais entrepris sur les contraintes globales à l'ECRC.

A COSYTEC à partir de 1991, j'ai donc créé des contraintes globales plus proches des types d'applications que nous développions dans le domaine de l'ordonnancement, du placement et des tournées. Cela a respectivement conduit aux contraintes suivantes :

[10] La motivation d'un traitement plus global est venue du manque de propagation présent dans le cas suivant : soient 3 variables X, Y et Z prenant des valeurs entières distinctes comprises entre 1 et 9 ; supposons de plus que $X<Z$ et $Y<Z$; un traitement local des contraintes précédentes ajuste la borne inférieure de la variable Z à la valeur 2, alors qu'un traitement prenant en compte le fait que X et Y sont distincts repousse la borne inférieure de Z à 3.

[11] Typiquement une contrainte locale mentionne un nombre restreint de variables.

– la contrainte *cumulative* [AGGOUN & BELDICEANU 92] limite la consommation de ressource instantanée d'un ensemble de tâches.
– la contrainte *diffn* [BELDICEANU & CONTEJEAN 94] impose le non-recoupement d'un ensemble de boîtes rectangulaires multidimensionnelles dont les côtés sont parallèles aux axes de l'espace de placement.
– la contrainte *cycle* [BELDICEANU & CONTEJEAN 94] impose le partitionnement d'un graphe orienté par un ensemble de circuits disjoints de manière à ce que chaque nœud du graphe appartienne à un circuit et un seul.

Chez ILOG une démarche similaire a conduit au développement de bibliothèques de contraintes spécialisées dans le domaine de l'ordonnancement [LE PAPE 94] et des tournées [SHAW 98]. A l'occasion du développement de ces contraintes « applicatives » on s'est vite aperçu qu'il était judicieux de réutiliser et d'adapter certains algorithmes développés par la communauté recherche opérationnelle. Cela a été particulièrement fait dans le domaine de l'ordonnancement avec la reprise des travaux de Jacques Carlier et d'Eric Pinson dans le domaine du disjonctif [CARLIER & PINSON 90] et par les nombreux travaux de Philippe Baptiste (`http://www.hds.utc.fr/~baptiste/`) dans les domaines des contraintes [BAPTISTE, LE PAPE & NUIJTEN 01] et de la complexité.

A partir de 1994, un travail de fond a également été mené chez ILOG par Jean-Charles Régin sur la catégorie des contraintes globales qui imposent le fait d'utiliser plus ou moins les valeurs d'une certaine façon. Ces contraintes se retrouvent dans un grand nombre de problèmes d'affectation. Pour un ensemble de variables données, la contrainte *alldifferent* [RÉGIN 94] impose l'utilisation de valeurs distinctes tandis que la contrainte *global_cardinality* [RÉGIN 99a] demande un nombre minimum et maximum d'occurrences pour chaque valeur potentielle d'une variable. Quant à la contrainte *symmetric_alldiff* [RÉGIN 99b], elle impose une permutation dont tous les cycles comportent deux sommets. Pour les contraintes précédentes, Jean-Charles Régin a développé des algorithmes produisant un élagage complet, souvent basés sur des algorithmes de flot [AHUJA, MAGNANTI & ORLIN 93]. Plus récemment, nous avons également contribué à cette ligne de recherche en proposant des algorithmes de filtrage pour la contrainte *nvalue* [BELDICEANU 01b] qui impose une restriction sur le nombre total de valeurs différentes que l'on affecte à un ensemble de variables.

A partir de 1997 Yves Caseau et son équipe se sont intéressés plus particulièrement aux approches hybrides [CASEAU, LABURTHE & SILVERSTEIN 99] combinant les contraintes et les méta-heuristiques. Plus récemment, dans le même ordre d'idée, on peut noter le développement des contraintes globales dans un contexte uniquement lié aux méta-heuristiques. C'est par exemple le cas des travaux de Alexander Nareyek [NAREYEK 01]

ainsi que de ceux de Pascal Van Hentenryck [MICHEL & VAN HENTENRYCK 02]. L'une des idées de cette approche consiste à remplacer les mouvements orientés problèmes par des mouvements orientés contraintes globales. Cela signifie qu'au lieu d'avoir une fonction de coût mesurant le degré de violation d'un problème dans son entier, on associe à chaque contrainte globale une fonction de coût indiquant son degré de violation. On peut alors, pour chaque contrainte globale, étudier un algorithme incrémental s'appuyant sur la structure de la contrainte globale en question.

Vers la fin des années 90, la programmation par contraintes restait une discipline jeune peu acceptée dans le milieu traditionnel de la recherche opérationnelle. Afin de conquérir ce marché et de posséder une offre complète dans le domaine de l'optimisation, ILOG décida donc de racheter la société dominante dans le domaine de la programmation linéaire. Soulignons que la programmation linéaire et la programmation par contraintes se complètent assez bien dans le domaine de la planification. En effet, on utilise souvent la programmation linéaire pour résoudre le problème de planification à long terme (quels produits faut-il produire et en quelles quantités). De son côté, la programmation par contraintes résout le problème de planification à court terme en tenant compte des contraintes opérationnelles liées aux caractéristiques des ressources utilisées.

Ce rachat de Cplex par ILOG à la fin des années 90 a contribué à développer la coopération de résolveurs entre, d'une part les domaines finis et d'autre part la programmation linéaire [BOCKMAYR & KASPER 98]. Cela a finalement abouti au langage de modélisation OPL [VAN HENTENRYCK 99] qui permet d'écrire des modèles se basant sur la programmation linéaire, sur les domaines finis ou sur une combinaison des deux.

2.2 Les systèmes contenant des contraintes globales et leur domaines d'applications

Il existe un nombre relativement important de systèmes de programmation par contraintes contenant des contraintes globales. Cette section les répertorie[12] en indiquant leur originalités respectives ainsi que leur domaine d'applications[13]. Pour chaque système de programmation par contraintes contenant des contraintes globales nous indiquons le ou les langages à travers lesquels ces contraintes sont accessibles. Notons qu'à l'heure actuelle CHIP, ILOG Solver et SICStus offrent un éventail plus ou moins complet de contraintes globales. Quant aux autres systèmes, ils offrent typiquement un nombre plus restreint de contraintes globales. Cet état de fait a également motivé nos recherches sur

[12] Nous ne considérons que les systèmes disponibles à l'heure actuelle (mars 2003).

[13] Nous utilisons pour cela des informations disponibles sur la toile.

le méta pour essayer de produire plus facilement un nombre important de contraintes globales. Les contraintes globales peuvent être disponibles sous les formes suivantes :

- une bibliothèque de contraintes,
- une extension d'un langage particulier,
- une bibliothèque de composants métiers.

Soulignons que certains systèmes sont disponibles sous plusieurs formes. A l'heure actuelle il n'existe pas de réponse définitive à la meilleure façon de présenter les contraintes globales. On peut cependant noter les problèmes suivants :

- pour un utilisateur novice il est bien souvent difficile de s'y retrouver dans un manuel décrivant un nombre important de contraintes,
- lorsque l'on présente les contraintes globales indépendamment de toute interprétation il est parfois difficile de faire le rapprochement avec un besoin concret,
- lorsque l'on présente les contraintes globales sous forme de composants métiers on limite arbitrairement leur utilisation à un cadre très précis,
- dans le cadre d'applications pratiques il manque bien souvent une contrainte particulière que l'on voudrait pour exprimer une condition spécifique.

Table 1. Systèmes contenant des contraintes globales.

Nom	Adresse sur la toile	Bibliothèque	Extension de langage	Composant métier
Bprolog	http://www.probp.com/	Non	Oui(Prolog)	Non
CHIP	http://www.cosytec.com/	Oui (C,C++)	Oui(Prolog)	Non
ECLAIR	(pas de site)	Oui (Claire)	Non	Non
ECLiPSe	http://www-icparc.doc.ic.ac.uk/eclipse/	Non	Oui(Prolog)	Non
FaCile	http://www.recherche.enac.fr/opti/facile/	Oui (OCaml)	Non	Non
ICEBERG	http://www.choco-constraints.net/download/ICEBERGuserguide0-5.pdf	Oui (Claire)	Non	Non
IF/PROLOG	http://www.ifcomputer.com/IFProlog/Constraints/home_en.html	Non	Oui(Prolog)	Non
ILOG Solver	http://www.ilog.com/	Oui(C++,Java)	Non	Non
ILOG Scheduler[14]	http://www.ilog.com/	Oui(C++)	Non	Oui(C++)
ILOG Configurator	http://www.ilog.com/	Non	Non	Oui(C++,Java)
Koalog	http://www.koalog.com/php/jcs.php	Oui(Java)	Non	Non
Mozart	http://www.mozart-oz.org/	Non	Non	Non
SICStus	http://www.sics.se/isl/sicstus/	Non	Oui(Prolog)	Non

[14] Ilog Scheduler est un peu difficile à classer : les objets principaux sont métiers (ordonnancement), mais en même temps il s'agit essentiellement d'une bibliothèque de contraintes.

Nous passons maintenant en revue les systèmes précédents.

2.2.1 Bprolog

Ce système est une extension de Prolog contenant un résolveur sur les domaines finis. Il comporte un nombre limité de contraintes globales telles que *alldifferent*, *element*, *cumulative* et *diffn*. Pour ces deux dernières contraintes il s'agit d'une version restreinte des contraintes que j'avais respectivement introduites dans [AGGOUN & BELDICEANU 92] et [BELDICEANU & CONTEJEAN 94]. Les applications industrielles mentionnées sur la toile concernent les problèmes de configuration et de routage. Ce système est commercialisé par la société Afany Software implantée aux États-Unis et en Asie.

2.2.2 CHIP

Avec Prolog III [COLMERAUER 90] et CLP(*R*) [JAFFAR, MICHAYLOW, STUCKEY & YAP 92] le système CHIP fut l'un des premiers systèmes de programmation logique avec contraintes. Originellement conçu à l'ECRC entre les années 1985 et 1990 il est maintenant développé par la société COSYTEC implantée en France. Ce système est disponible à travers les langages Prolog, C et C++. Son originalité provient du fait qu'il offre un nombre limité de contraintes globales possédant une grande variété de possibilités de paramétrages. Les principales contraintes globales de CHIP sont les contraintes *alldifferent* (problèmes d'affectation), *cycle* (problèmes de tournées), *diffn* (problèmes de placement géométrique), *sequence* (problèmes de réglementations), *cumulative* (problèmes d'ordonnancement avec capacité restreinte), *cumulative-trapèze* (problèmes d'ordonnancement avec consommation continue de ressource) et *prod_cons* (problèmes d'ordonnancement avec ressource non-renouvelables). Parmi ces contraintes, j'ai conçu et implémenté les contraintes *cycle*, *diffn*, *sequence* et *cumulative*. De plus j'ai encadré deux thèses concernant respectivement la contrainte *cycle* [BOURREAU 99] et la contrainte *cumulative-trapèze* [PODER 02]. Le système CHIP est vendu en tant que tel ; il est également utilisé pour faire des applications dans les domaines de la production (PSA Peugeot Citroën, BASF), dans le domaine de l'audiovisuel (CANAL+, Radio France Outremer) ou dans le domaine de l'énergie (EDF: évacuation du combustible irradié).

2.2.3 ECLAIR

ECLAIR [PLATON 03] est une bibliothèque de contraintes au-dessus de CLAIRE contenant quelque contraintes globales. Elle fut développée et utilisée en interne chez Thomson-CSF puis chez Thales. Ces sociétés choisirent un système dont ils maîtrisaient complètement le code source ce qui n'est pas le

cas des systèmes commerciaux classiques. En effet, cette approche est motivée par le fait que ces sociétés développent des applications embarquées dans le domaine du militaire. Ces applications ont un long cycle de vie et tous les logiciels utilisés doivent répondre à des normes strictes. En ce qui concerne les contraintes globales, ECLAIR contient les contraintes *alldifferent*, *element*, *minimum*, *maximum* et *nbchanges* (dans une séquence de variables nombre de changements de valeurs entre deux variables consécutives[15]). L'algorithme de filtrage de cette dernière contrainte est basé sur un algorithme publié dans [BELDICEANU & CARLSSON 01a].

2.2.4 ECLiPSe

ECLiPSe est un système de programmation par contraintes dérivé initialement de CHIP et utilisé et maintenu par le centre de recherche IC-PARC (`http://www-icparc.doc.ic.ac.uk/`) et sa start-up Parc Technologies Limited implantée à Londres. Cette société travaille dans les domaines de l'aérien ainsi qu'auprès de fournisseurs d'accès Internet. Elle développe des solutions basées sur des approches hybrides combinant la programmation par contraintes, les méta-heuristiques et la programmation linéaire.

2.2.5 FaCile

FaCile est une bibliothèque de contraintes sur les domaines finis développée en OCaml par Nicolas Barnier et Pascal Brisset au laboratoire d'optimisation globale de l'Ecole Nationale de l'Aviation Civile à Toulouse. Elle contient les contraintes globales *alldifferent*, *element*, *global_cardinality* et *sort*. Cette bibliothèque a par exemple été utilisée dans le domaine de la gestion du trafic aérien [BARNIER & BRISSET 00] ainsi que pour l'enseignement.

2.2.6 ICEBERG

ICEBERG est une bibliothèque dédiée aux contraintes globales développée au-dessus du système de programmation par contraintes CHOCO [LABURTHE 00]. Cette bibliothèque est conçue au sein du laboratoire de recherche e-lab du groupe Bouygues. L'objet de ce laboratoire est de traiter des projets de recherche et développement pouvant rapidement aboutir à des applications opérationnelles dans des domaines clefs des différents métiers traditionnels du groupe Bouygues : audiovisuel avec TF1, téléphonie avec Bouygues Télécom, construction et chantier avec Bouygues.

---

[15] Par exemple, la contrainte nbchanges(2,[6,6,3,1,1,1,1]) est vérifiée car la séquence 6 6 3 1 1 1 1 comporte exactement deux changements de valeurs.

2.2.7 IF/PROLOG

IF/PROLOG est un système Prolog offrant des extensions dans le domaine des contraintes. Il est commercialisé par la société IFComputer (http://www.ifcomputer.com/) qui est rattachée à SIEMENS. IFComputer possède des filiales en Allemagne et au Japon. Le manuel de référence concernant la partie contrainte (http://www.ifcomputer.com/IFProlog/Manuals/v5.3/cons.pdf) mentionne les contraintes globales *alldifferent*, *element*, *cardinality*, *cumulative*, *diffn*, *minimum* et *maximum*. Pour la contrainte *diffn* elle offre plusieurs variantes que nous avions proposées dans [BELDICEANU & CONTEJEAN 94].

2.2.8 ILOG Solver, ILOG Scheduler, ILOG Configurator

ILOG Solver [PUGET 94], Scheduler [LE PAPE 94], [NUIJTEN & LE PAPE 98], Dispatcher et Configurator sont des produits développés et commercialisés par la société française ILOG (http://www.ilog.com/) basée à la fois en France et aux Etats-Unis. Ces logiciels sont fournis sous la forme de bibliothèques C++ ou Java pour certaines. Le point fort de cette offre vient du fait qu'ILOG propose une panoplie complète d'outils dans le domaine de l'optimisation. En dehors de la programmation par contraintes on peut mentionner CPLEX (programmation linéaire) ainsi qu'un outil de modélisation OPL permettant de combiner la programmation par contraintes et la programmation linéaire. Du point de vue des contraintes globales certaines sont directement disponibles en tant que telles tandis que d'autre sont «camouflées» sous forme de composants métiers destinés à un contexte spécifique. C'est le cas des bibliothèques Scheduler et Configurator. La diversité des technologies offertes par ILOG est également un frein à la diffusion de ces outils auprès d'utilisateurs devant maîtriser trop de concepts différents. Cela a conduit à l'introduction des composant métiers précédemment mentionnés.

2.2.9 Koalog

Koalog est une bibliothèque de contraintes sur les domaines finis en Java développée et vendue par la jeune société française Koalog (http://www.koalog.com/php/index.php). Les applications mentionnées par cette société concernent les domaines de l'ordonnancement et de la configuration. A l'heure actuelle cette bibliothèque comporte, au niveau des contraintes globales, les contraintes *alldifferent*, *circuit*, *sort* et *inverse*.

2.2.10 Mozart

Mozart est une plate-forme pour développer des applications intelligentes distribuées. Cette plate-forme est disponible sous la forme d'un logiciel libre.

Mozart est basé sur le langage Oz [SMOLKA 95] dont le concepteur principal est Gert Smolka. Ce langage combine les aspects distribués, la concurrence, les objets et les contraintes. Il fut développé par l'université de Saarbrücken, par le Swedish Institute of Computer Science et par l'université de Louvain. La partie contrainte [SMOLKA 96] comporte les contraintes globales *alldifferent*, *element*, *distinct2* (non-recoupement de rectangles) et *cumulative*.

2.2.11 SICStus

SICStus est un système Prolog développé depuis bientôt 15 ans par le centre de recherche SICS (`http://www.sics.se/`) établi en Suède. Il comporte un résolveur sur les domaines finis qui contient un nombre assez important de contraintes globales que j'ai développées en partie. Mes contributions exactes [BELDICEANU 01b], [ÅGREN, SZEREDI, BELDICEANU & CARLSSON 02], [BELDICEANU & CARLSSON 01a], [BELDICEANU & CARLSSON 01b], [BELDICEANU & CARLSSON 02], [CARLSSON & BELDICEANU 02a], [CARLSSON & BELDICEANU 02b], [BELDICEANU, CARLSSON & THIEL 03] à ce système seront détaillées dans la dernière partie de ce mémoire. SICStus est d'une part utilisé pour l'enseignement et d'autre part pour des applications. En ce qui concerne les contraintes il est, à notre connaissance, utilisé[16] dans les domaines des télécommunications chez ERICSSON (`http://www.ericsson.com/se/`), dans le domaine de la configuration chez Tacton (`http://www.tacton.com/`) ou dans le domaine des biotechnologies chez PyroSequencingAB (`http://www.pyrosequencing.com/`) et à l'INRIA.

2.3 Bibliographie thématique sur les contraintes globales

Cette section fournit une bibliographie thématique comportant les principaux articles concernant les contraintes globales. Nous avons organisé cette bibliographie dans les thèmes suivants :

- les contraintes globales proprement dites,
- les contraintes globales avec fonction de coût,
- les contraintes globales et la programmation linéaire,
- la relaxation de contraintes globales,
- les conjonctions de contraintes globales,
- les contraintes globales et les méta-heuristiques.

[16] Il s'agit d'applications dont nous sommes au courant. Pour les deux derniers exemples nous avons contribué à leur développement. Nous ne connaissons cependant pas la liste des clients de SICStus.

2.3.1 Contraintes globales

Pour chaque contrainte nous donnons son nom, sa définition, sa finalité ainsi que les articles la concernant.

Alldifferent
La contrainte *alldifferent* impose que les variables d'une collection de variables soient deux à deux distinctes. Cette contrainte fut introduite dans ALICE [LAURIÈRE 76] par Jean Louis Laurière sous une forme légèrement différente (on imposait une relation injective entre un ensemble de variables et les valeurs que celles-ci peuvent prendre). Le premier algorithme de filtrage complet pour cette contrainte à été découvert de manière indépendante par Marie Christine Costa [COSTA 94] et Jean Charles Régin [RÉGIN 94]. Cet algorithme est basé sur un corollaire de Berge qui caractérise les arêtes d'un graphe appartenant à un couplage maximum mais non à tous [BERGE 70, page 120]. Par la suite des algorithmes de consistance sur les bornes des variables ont été proposés par [PUGET 98] et [MEHLHORN & THIEL 00]. D'un point de vue pratique la contrainte *alldifferent* se rencontre dans la plupart des problèmes.

All_pair_diff
La contrainte *all_pair_diff* impose que les paires de variables d'une collection de paires de variables soient deux à deux distinctes. Deux paires (u_1,u_2) et (v_1,v_2) sont distinctes si $u_1 \neq v_1$ ou si $u_2 \neq v_2$. Elle fut introduite dans [BELDICEANU 90] pour traiter des problèmes de blocs (triplets de Steiner, triplets de Kirkman, carrés latin orthogonaux). L'algorithme de filtrage proposé à l'époque tenait également compte du fait que certaines variables doivent prendre des valeurs distinctes.

Among et **global cardinality**
Dans sa version de base cette contrainte fut respectivement introduite[17] dans CHIP [BELDICEANU & CONTEJEAN 94] et Ilog Solver [RÉGIN 96]. Elle impose une fourchette sur le nombre d'occurrence de différentes valeurs données parmi une collection de variables. Dans [RÉGIN 96], Jean Charles Régin propose un algorithme d'élagage complet pour cette contrainte. Cette contrainte est utile dans certains problèmes d'affectation.

[17] Elle existait déjà dans CHARME et portait la dénomination de *distribute* mais nous ne connaissons pas d'article la définissant.

Cardinality

L'opérateur[18] de cardinalité fut introduit dans [VAN HENTENRYCK & DEVILLE 91]. Il prend comme argument une variable domaine *Compte* et une liste de contraintes *ListeCtr* et impose qu'exactement *Compte* contraintes de la liste *ListeCtr* soient satisfaites. Son traitement est basé sur le fait que chaque contrainte de la liste *ListeCtr* détecte si elle est toujours satisfaite ou si elle n'est à coup sûr pas satisfaite. En fonction de cela on ajuste les bornes de la variable *Compte* et l'on propage éventuellement par rapport aux contraintes dont l'état n'est pas encore déterminé. L'exemple suivant illustre ce traitement. Soit l'appel cardinality(C,[$X_1+4\leq X_2$, $X_2+1\leq X_1$, $Y_1+2\leq Y_2$, $Y_2+4\leq Y_1$]) dans lequel les domaines des variables C, X_1, Y_1, X_2 et Y_2 sont définis comme suit : dom(C)={1,2,3,4}, dom(X_1)={1,2}, dom(Y_1)={2,3}, dom(X_2)={3,4,5,6,7,8,9} et dom(Y_2)={0,1,2,3}. Les trois contraintes $X_2+1\leq X_1$, $Y_1+2\leq Y_2$, $Y_2+4\leq Y_1$ étant forcément fausses, on ajuste le maximum de la variable C à 1. Finalement comme C est égale à 1 et qu'une seule contrainte parmi les quatre peut être vérifiée, on impose la contrainte $X_1+4\leq X_2$, ce qui conduit à ajuster le minimum de la variable X_2 à la valeur 5. L'exemple que nous venons de donner illustre une utilisation de l'opérateur de cardinalité pour modéliser le fait que deux rectangles R_1 et R_2 de tailles respectives 4,2 et 1,4 ne peuvent se couper. Les quatre contraintes mentionnées dans la liste de contraintes [$X_1+4\leq X_2$, $X_2+1\leq X_1$, $Y_1+2\leq Y_2$, $Y_2+4\leq Y_1$] correspondent respectivement au fait que R_1 est situé à gauche de R_2, que R_1 est situé à droite de R_2, que R_1 est situé en dessous de R_2 et que R_1 est situé au-dessus de R_2.

Cumulative

La contrainte *cumulative* fut introduite dans l'article [AGGOUN & BELDICEANU 92] pour traiter des problèmes d'ordonnancement impliquant une ressource de capacité limitée. La contrainte *cumulative* a la forme suivante cumulative([$O_1,\ldots,O_n$],[$D_1,\ldots,D_n$],[$R_1,\ldots,R_n$],*Limit*) où [$O_1,\ldots,O_n$], [$D_1,\ldots,D_n$] et [$R_1,\ldots,R_n$] sont des listes de variables domaine et où *Limit* est une variable domaine. La contrainte impose que $Limit=\max\left(\sum_{j|O_j\leq i\leq O_j+D_j-1} R_j\right)$.

Elle s'interprète comme le fait que *Limit* est égal au pic maximum de ressources consommées par les différentes tâches. Les variables O_i, D_i et R_i représentent respectivement l'origine, la durée et la quantité de ressources

[18] On utilise ici le terme *opérateur* au lieu du terme *contrainte* pour marquer le fait suivant: un opérateur accepte comme arguments des contraintes ou des variables tandis qu'une contrainte n'accepte que des variables.

consommées par la ième tâche. Chaque tâche i monopolise de manière instantanée une certaine quantité R_i de ressources pendant toute la durée de son exécution. La contrainte *cumulative* intervient dans la plupart des problèmes d'ordonnancement que l'on rencontre dans la pratique. L'article [CASEAU & LABURTHE 96b] propose des algorithmes de filtrage basés sur la notion d'intervalle de tâches (apparentée aux travaux de [ERSCHLER & LOPEZ 90]) et de partie obligatoire [LAHRICHI 82].

Cumulatives
La contrainte *cumulatives* fut introduite récemment dans l'article [BELDICEANU & CARLSSON 02]. Elle généralise la contrainte *cumulative* en autorisant des consommations de ressource négative et en introduisant plusieurs ressources. Cette dernière caractéristique se rencontre dans un grand nombre de problèmes d'ordonnancement pratiques dans lesquels on n'a pas encore affecté les tâches à une ressource particulière. Le principal algorithme décrit dans [BELDICEANU & CARLSSON 02] repose sur une méthode de balayage.

Cumulative-trapèze
La contrainte *cumulative-trapèze* fut introduite dans CHIP [PODER 02] afin de modéliser les problèmes d'ordonnancement dans lesquels la consommation des tâches en ressources est variable dans le temps. Chaque tâche est représentée par une suite de sous-tâches trapézoïdales contiguës dont les hauteurs et la base sont variables. La contrainte impose à ne pas dépasser un plafond de ressource maximum. Un des algorithmes utilisé par la contrainte *cumulative-trapèze* est décrit dans [PODER, BELDICEANU & SANLAVILLE 02]. Il s'agit du calcul de la partie obligatoire[19] d'une tâche.

Cycle
La contrainte *cycle* fut introduite dans CHIP [BELDICEANU & CONTEJEAN 94] pour des problèmes de partitionnement des nœuds d'un graphe orienté par des circuits. Dans sa forme de base cycle(N,[V_1,…,V_m]), N correspond à une variable domaine et [V_1,…,V_m] à une liste de variables domaine. Elle impose que [V_1,…,V_m] soit une permutation contenant N cycles distincts. Par exemple, la contrainte cycle(3,[3,4,1,2,5]) est vérifiée car la permutation <3,4,1,2,5> comporte effectivement trois cycles. Dans son interprétation en terme de couverture de graphe orienté par des circuits le domaine initial de chaque variable V_i correspond aux successeurs potentiels du nœud i dans le graphe orienté que l'on considère. La contrainte *cycle* fut étendue dans la

[19] La partie obligatoire d'une tâche correspond à l'intersection de toutes les instances réalisables de la tâche.

thèse d'Eric Bourreau [BOURREAU 99] afin de prendre en compte un grand nombre de contraintes opérationnelles sur les circuits que l'on veut finalement obtenir. Un cas particulier de la contrainte cycle est la contrainte *circuit*. Celle-ci impose une permutation ne contenant qu'un seul cycle.

Diffn

La contrainte *diffn* fut introduite dans CHIP [BELDICEANU & CONTEJEAN 94] pour des problèmes de placement et d'ordonnancement disjonctif avec choix de machine. Dans sa forme de base la contrainte *diffn* prend une liste de boîtes rectangulaires multidimensionnelles (dont les côtés sont parallèles aux axes de l'espace de placement) et impose que ces boîtes ne se coupent pas. Chaque boîte est définie par les coordonnées de son origine et ses tailles. Plus précisément la contrainte

$$\text{diffn}([[O_{11},\ldots,O_{1n},T_{11},\ldots,T_{1n}],\ldots,[O_{m1},\ldots,O_{mn},T_{m1},\ldots,T_{mn}]])$$

impose la condition suivante :

$$\forall i\in[1,m], \forall j\in[1,m]\ j\neq i,\ \exists k\in[1,n] \text{ tel que } O_{ik}\geq O_{jk}+T_{jk} \vee O_{jk}\geq O_{ik}+T_{ik}.$$

Pour chaque paire de boîtes distinctes B_i et B_j il existe au moins une dimension k dans laquelle B_i est située après ou avant B_j. Un algorithme de filtrage applicable à la contrainte *diffn* est décrit dans [BELDICEANU, GUO & THIEL 01].

Disjoint2

La contrainte *disjoint2* est une version restreinte de la contrainte *diffn* dans laquelle on ne considère que des rectangles. Un algorithme de filtrage pour cette contrainte est décrit dans [BELDICEANU & CARLSSON 01b].

Element

La contrainte *element* introduite dans [VAN HENTENRYCK & CARILLON 88] lie deux variables *Index* et *Var* par l'intermédiaire d'une liste de variables *ListeVar*. Plus précisément la contrainte element(*Index*,*ListeVar*,*Var*) impose que la variable *Var* soit égale à la variable située à la *Index*ième position de la liste *ListeVar*. Lorsque toutes les variables de *ListeVar* sont fixées la contrainte *element* est utilisée pour modéliser les types de problèmes suivants :

- la définition en extension d'une fonction, et en particulier la définition de fonctions de coûts,
- le lien entre deux attributs d'un objet ; par exemple, dans un problème d'ordonnancement avec choix de machine, on relie la machine à laquelle

est affectée une tâche à sa durée effective par l'intermédiaire d'une contrainte *element*[20].

La contrainte *element* permet également d'exprimer directement une disjonction entre plusieurs égalités. Ainsi l'expression element(X,[T_1,T_2,T_3],Y) dans laquelle X peut prendre les valeurs 1,2 ou 3 impose la condition $Y=T_1 \vee Y=T_2 \vee Y=T_3$. De même que la contrainte *alldifferent*, elle se retrouve dans la quasi-totalité des systèmes de programmation par contraintes. À notre connaissance il n'existe pas d'article décrivant un algorithme de filtrage pour la contrainte *element*.

Inverse

La contrainte inverse([X_1,…,X_n],[Y_1,…,Y_n]) impose l'implication suivante :

$$X_i=j \Leftrightarrow Y_j=i\ .$$

Elle peut s'interpréter comme le fait que $\langle Y_1,\ldots,Y_n\rangle$ soit la permutation inverse de la permutation $\langle X_1,\ldots,X_n\rangle$. Bien que cette contrainte soit présente dans un grand nombre de systèmes de programmation par contraintes, nous ne connaissons pas d'article lui étant consacrée. La contrainte *inverse* sert pour écrire certaines heuristiques. Ainsi dans le cas de problèmes où l'on a une bijection entre un ensemble de variables et un ensemble de valeurs on peut vouloir créer une variable[21] pour chaque valeur afin d'exprimer le fait que l'on veut choisir l'alternative possédant le moins de branches.

≤lex

Étant donnés deux vecteurs de variables domaine $\langle X_0,\ldots,X_{n-1}\rangle$ et $\langle Y_0,\ldots,Y_{n-1}\rangle$, $\langle X_0,\ldots,X_{n-1}\rangle \leq_{\text{lex}} \langle Y_0,\ldots,Y_{n-1}\rangle$ représente la contrainte d'ordre lexicographique entre les deux vecteurs précédents. Cette contrainte est vérifiée si et seulement si $n=0$, ou bien $X_0<Y_0$, ou bien $X_0=Y_0$ et $\langle X_1,\ldots,X_{n-1}\rangle \leq_{\text{lex}} \langle Y_1,\ldots,Y_{n-1}\rangle$. A l'origine cette contrainte fut introduite dans CHIP. Le premier algorithme de filtrage détectant toutes les valeurs inconsistantes fut décrit dans l'article [FRISCH, HNICH, KIZILTAN, MIGUEL & WALSH 02]. L'article [CARLSSON & BELDICEANU 02a] améliore cet algorithme en détectant en plus certains cas où la contrainte est forcément satisfaite. Cette contrainte est utilisée dans le cas de matrices de variables dont on veut ordonner lexicographiquement les colonnes ou les lignes afin d'éliminer des solutions symétriques. Une autre contrainte d'ordre pour des

---

[20] Supposons que la durée d'une tâche T soit égale à 10 si on l'affecte à la machine 1 et 15 si on l'affecte à la machine 2. On relie la variable M (machine à laquelle est affectée la tâche T) à la variable D (durée de la tâche T) par l'intermédiaire de la contrainte element(M,[10,15],D).

[21] Le domaine de la variable associée à la valeur v est constitué par les indices des variables pouvant prendre la valeur v.

problèmes similaires a récemment été introduite dans [FRISCH, HNICH, KIZILTAN, MIGUEL & WALSH 03].

Lexchain

La contrainte *lexchain* impose la condition $X_0 \leq_{lex} X_1 \leq_{lex} \ldots \leq_{lex} X_{m-1}$ dans laquelle X_0, $X_1,\ldots,X_{m-1}$ correspondent à des vecteurs de variables domaine possédant un même nombre de composantes. Bien que l'on puisse la décomposer dans $m-1$ contraintes d'ordre lexicographique, elle fut introduite dans [CARLSSON & BELDICEANU 02b] afin de fournir un algorithme de filtrage détectant toutes les valeurs inconsistantes. Son contexte d'utilisation est le même que celui de la contrainte d'ordre lexicographique.

Minimum

La contrainte minimum(M,[$X_1,\ldots,X_n$]) impose que la variable domaine M soit égale au minimum de la liste de variables domaine [$X_1,\ldots,X_n$]. Elle est présente dans un grand nombre de systèmes et nous ne pouvons guère situer son origine. Un algorithme de filtrage pour cette contrainte ainsi que pour une généralisation (minimum de rang r) est décrit dans [BELDICEANU 01b]. Cette contrainte est utile pour exprimer certain type de contraintes de précédence dans le cadre de problèmes d'ordonnancement.

Nvalue

La contrainte nvalue(D,[$X_1,\ldots,X_n$]) impose que la variable domaine D soit égale au nombre de valeurs distinctes prises par les variables domaine $X_1,\ldots,X_n$. Un premier algorithme de filtrage est décrit dans [BELDICEANU 01b] et un deuxième algorithme dans [BELDICEANU, CARLSSON & THIEL 02]. Ce dernier détecte toutes les valeurs inconsistantes lorsque le domaine de chaque variable n'est constitué que d'un seul intervalle de valeurs consécutives (pas de trous dans les domaines). Cette contrainte intervient dans des problèmes ou l'on veut limiter le nombre maximum de valeurs utilisées par un ensemble de variables. Par exemple, dans un problème d'ordonnancement on veut limiter le nombre de machines distinctes effectivement utilisées. De la même manière, dans un problème de tournées on pourra également vouloir limiter le nombre de villes visitées (tout en sachant que l'on peut visiter plus d'un client dans une ville donnée).

Sort

Étant donnés deux listes de variables domaine [$X_1,\ldots,X_n$] et [$S_1,\ldots,S_n$], la contrainte sort([$X_1,\ldots,X_n$],[$S_1,\ldots,S_n$]) impose que [$S_1,\ldots,S_n$] soit triée par ordre faiblement croissant et que [$X_1,\ldots,X_n$] soit obtenue par une permutation des éléments de [$S_1,\ldots,S_n$]. Cette contrainte est mentionnée pour la première fois dans [OLDER, SWINKELS & EMDEN 95] et un premier algorithme de

filtrage est décrit dans [GUERNALEC & COLMERAUER 97]. Un deuxième algorithme est présenté dans [MEHLHORN & THIEL 00]. Cette contrainte est utile pour résoudre des problèmes d'ordonnancement d'atelier. Une variante de cette contrainte fut introduite dans [ZHOU 97]. Dans cette variante une troisième liste de variables domaine représente la permutation faisant passer de $[X_1,\ldots,X_n]$ à $[S_1,\ldots,S_n]$.

Stretch

La contrainte *stretch* fut introduite par Gilles Pesant [PESANT 01] dans le cadre de problème d'établissement de planning dans le monde hospitalier. Cette contrainte restreint le nombre de fois que peut apparaître de manière consécutive une valeur donnée. Une fourchette est fixée pour chaque valeur. L'article de Pesant présente un ensemble de règles de propagation locales pour traiter cette contrainte.

Symmetric_alldifferent

La contrainte *symmetric_alldifferent* fut introduite par Jean-Charles Régin [RÉGIN 99a] dans le cadre de problèmes d'établissement de planning dans des tournois sportifs. Dans ces problèmes il s'agit de former des rencontres entre des équipes. Dans ce contexte chaque rencontre correspond à l'association de deux équipes données. Plus formellement, étant donnée une liste de variables domaine $[X_1,\ldots,X_n]$ la contrainte symmetric_alldifferent($[X_1,\ldots,X_n]$) impose que $<X_1,\ldots,X_n>$ soit une permutation ne comportant que des cycles de longueur 2. Par exemple la contrainte symmetric_alldifferent([6,5,4,3,2,1]) est vérifiée car <6,5,4,3,2,1> est une permutation comportant 3 cycles de longueur 2. L'algorithme de filtrage décrit dans [RÉGIN 99a] est basé sur un algorithme couplage.

2.3.2 Contraintes globales avec fonction de coût

Il s'agit de contraintes globales dans lesquelles on a :

- d'une part un ensemble de variables de décision,
- d'autre part une variable de coût ainsi que des éventuels paramètres supplémentaires (matrice de coût, pondérations associées aux valeurs, …) permettant de calculer le coût en fonction des valeurs prises par les variables de décision.

En règle générale la plupart des algorithmes associés à cette catégorie de contraintes globales correspondent à l'adaptation d'un algorithme de recherche opérationnelle (comme par exemple l'algorithme hongrois dans le cas du couplage pondéré). Voici la liste des principales contraintes globales avec fonction de coût :

- couplage pondéré [CASEAU & LABURTHE 97a], [SELLMANN 02],

– voyageur de commerce [CASEAU & LABURTHE 97b],
– affectation [FOCCACI, LODI & MILANO 99],
– extension de la contrainte *global_cardinality* [RÉGIN 99b],
– clique de taille maximale [FAHLE 02],
– exploitation des coûts réduits lors de l'énumération [MILANOE & HOEVE 02],
– contrainte sur la somme des poids associés aux valeurs distinctes [BELDICEANU, CARLSSON & THIEL 02].

2.3.3 Contraintes globales et programmation linéaire en nombres entiers

La programmation linéaire en nombres entiers [SCHRIJVER 86] a bien vite reconnu l'intérêt des contraintes globales. En effet, chaque contrainte globale capture une sous-structure dont on peut tirer partie pour faire des traitements particuliers. Depuis bien longtemps les résolveurs de programmation linéaire essaient de reconnaître dans une phase de prétraitement certaines sous-structures particulières pour lesquelles ils utilisent des méthodes spécifiques. Le fait d'utiliser des contraintes globales dans le cadre de la programmation linéaire permet de donner explicitement les sous-structures présentes dans le modèle initial et évite ainsi d'essayer de les retrouver à partir de la matrice associée au modèle [ALTHAUS, BOCKMAYR, ELF, KASPER, JÜNGER & MEHLHORN 02]. C'est dans ce contexte que la communauté programmation linéaire a commencé à étudier comment exploiter la structure de certaines contraintes globales venant de la programmation par contraintes. Voici les principaux articles concernant ce sujet :
– coopération contraintes globales-programmation linéaire [BOCKMAYR & KASPER 98], [OTTOSSON, THORSTEINSSON & HOOKER 99],
– coupes pour certaines contraintes globales [REFALO 00],
– relaxation pour la contrainte *cumulative* [HOOKER & YAN 02],
– contrainte de somme avec choix [YUNES 02].

Finalement soulignons que l'article [ALTHAUS, BOCKMAYR, ELF, KASPER, JÜNGER & MEHLHORN 02] et la bibliothèque lui correspondant (`http://www.mpi-sb.mpg.de/SCIL/`) sont largement basés sur le principe de classification des contraintes globales décrit dans [BELDICEANU 00c]. En effet, cette bibliothèque fournit dans le cadre de la programmation linéaire un ensemble d'abstractions basées sur un graphe orienté dans lequel on associe une variable 0-1 à chaque arc. Chaque abstraction correspond à une propriété (e.g. chemin, circuit, …) que l'on demande sur le sous graphe ne contenant que les arcs associés aux variables prenant la valeur 1.

2.3.4 Relaxation de contraintes globales

Dans un grand nombre d'applications, il est bien rare dans la pratique que l'on puisse obtenir une solution satisfaisant toutes les contraintes initialement imposées. Cela a suscité un ensemble de travaux dans le domaine de la relaxation de contraintes qui se sont initialement placés dans un cadre général où l'on ne tient pas compte de la structure des contraintes considérées : on veut par exemple satisfaire un maximum de contraintes indépendamment de la structure de ces contraintes. Cette approche générale présente les deux inconvénients suivants :

- les algorithmes obtenus ne tirant pas partie de la structure particulière des contraintes sont parfois inefficaces,
- dans la pratique on souhaite souvent relaxer les contraintes d'une manière spécifique ; ainsi, dans un problème d'ordonnancement avec contraintes disjonctives on voudra par exemple maximiser le nombre de tâches effectivement traitées ; dans un problème à contrainte cumulative on désirera minimiser les dépassements de consommation de ressource.

C'est dans ce contexte que l'on a développé pour certaines contraintes globales, des algorithmes de filtrage permettant de propager même dans le cas où ces contraintes doivent être partiellement relaxées. Voici les principaux articles concernant ce domaine :

- relaxation dans le domaine de l'ordonnancement [BAPTISTE, LE PAPE & PERIDY 98],
- contraintes sur des séquences de variables [BELDICEANU & CARLSSON 01a],
- algorithmes pour deux types de relaxation de la contrainte *alldifferent* [PETIT, RÉGIN & BESSIÈRE 01],
- algorithmes pour satisfaire le nombre maximum de contraintes dans le cadre de contraintes propageant sur les valeurs minimum et maximum des variables [PETIT, RÉGIN & BESSIÈRE 02].

2.3.5 Conjonction de contraintes globales

Dans certains cas on a établi des algorithmes de filtrage traitant directement la conjonction de deux types de contraintes. C'est par exemple le cas pour les deux articles suivants :

- combine un ensemble de contraintes de précédence avec plusieurs contraintes cumulatives [BELDICEANU, BOURREAU, RIVREAU & SIMONIS 96],
- combine une contrainte de somme sur un terme linéaire avec des inégalités [RÉGIN & RUEHER 99].

2.3.6 Contraintes globales et méta-heuristiques

Assez récemment on s'est intéressé aux contraintes globales dans le cadre des méta-heuristiques. Dans ce contexte, ce qui a retenu l'attention, ce ne sont pas les algorithmes de filtrage associés aux contraintes globales mais plutôt le fait que les contraintes globales puissent être vues comme des briques de bases permettant de modéliser différents aspects d'un problème. En effet, dans le cadre des méta-heuristiques on s'était jusqu'à maintenant intéressé à des voisinages orientés problèmes (job-shop, flow-shop, voyageur de commerce, …). Il est cependant difficile dans la pratique de redéfinir un voisinage tenant compte efficacement d'un grand nombre de contraintes hétérogènes. C'est pour cette raison que l'on a essayé d'associer un voisinage au niveau de chaque contrainte globale. Dans ce cadre il est maintenant possible d'essayer de trouver les meilleurs algorithmes incrémentaux permettant l'exploration du voisinage associé à chaque contrainte globale. Bien entendu ces algorithmes tireront partie de la structure particulière de chaque contrainte globale afin d'être plus efficace. Les travaux suivants font partie de cette ligne de recherche qui n'en est qu'à ces balbutiements :

– [NAREYEK 01],
– [MICHEL & VAN HENTENRYCK 02].

3 Contraintes globales et algorithmes

Ce chapitre dégage nos contributions dans le domaine des contraintes globales en se focalisant sur les techniques de résolutions utilisées dans les algorithmes de filtrage. Rappelons que nous nous limitons au cadre des domaines finis dans lequel se situent les contraintes globales.

Replaçons-nous d'abord dans le contexte historique de la fin des années 80 où se situe la genèse des contraintes globales. Dans les systèmes de programmation par contraintes de l'époque, l'immense majorité des contraintes correspondait à des contraintes numériques, à la contrainte de diségalité et à la contrainte *element*. Lorsque au tout début des années 90 les contraintes ont commencé à vraiment être utilisées dans l'industrie on a très tôt constaté les problèmes suivants :

- il existe souvent un fossé sémantique significatif entre les contraintes du problème que l'on cherche à résoudre et les contraintes effectivement disponibles dans un système de programmation par contraintes.
- dans un grand nombre de problèmes pratiques les contraintes sont subordonnées à la présence de certaines conditions. Le fait d'avoir des contraintes conditionnelles oblige à retarder leur prise en compte à la phase d'énumération ce qui pose des problèmes d'efficacité.
- lorsque l'on exprime une contrainte par un ensemble de contraintes plus élémentaires les techniques de propagation ne sont souvent pas assez efficaces pour prendre directement en compte l'interaction entre ces contraintes élémentaires.

Ce contexte nous a amené à introduire les contraintes globales. Nous caractérisons une contrainte globale par les trois traits suivants :

- chaque contrainte globale correspond à une notion centrale (permutation, tri, non-recoupement, ordonnancement cumulatif, …). Cette notion peut souvent s'interpréter de multiples façons lorsqu'on la confronte à des problèmes pratiques. Un exemple particulièrement frappant de cette multiplicité d'interprétations concerne la contrainte *cycle* [BELDICEANU & CONTEJEAN 94] qui peut se voir comme le nombre de cycles d'une permutation ou le fait que l'on veuille partitionner un graphe orienté par un ensemble de circuits de manière à ce que chaque nœud appartienne à un circuit et un seul.
- les paramètres d'une contrainte globale correspondent à des ensembles de variables dans lesquels tous les paramètres peuvent être des entrées ou des sorties. Ce point est important car il élargit le champ d'application de la

contrainte[22] et permet parfois d'exprimer une contrainte relaxée. Bien souvent il est judicieux pour des raisons de clarté de considérer que l'on a une collection d'objets dont les attributs correspondent à des variables. Par exemple dans le cas de la contrainte *cumulative* [AGGOUN & BELDICEANU 92] on peut dire que l'on a une collection de tâches, chaque tâche possédant un attribut origine, un attribut durée et un attribut quantité de ressources consommées.

– chaque contrainte globale utilise des algorithmes et des structures de données adaptées de manière à limiter la consommation mémoire et à détecter au plus vite pour chacune de ses variables les valeurs conduisant à coup sûr à un échec de la contrainte.

En dehors du fait que nous avons été l'un des premiers [BELDICEANU 90] à dégager clairement la notion de contrainte globale nos principales contributions dans ce domaine sont de trois ordres :

– il a d'abord bien fallu trouver les abstractions qui sont récurrentes à un grand nombre de problèmes [AGGOUN & BELDICEANU 92], [BELDICEANU & CONTEJEAN 94]. De manière paradoxale ce n'est pas une tâche aisée. En effet, après avoir trouvé une abstraction, celle-ci s'impose naturellement comme une évidence. Cependant cette manière de penser va à l'encontre des habitudes correspondant à voir chaque problème combinatoire comme un tout.
– après avoir trouvé une abstraction il faut chercher un algorithme de filtrage efficace permettant d'éliminer les valeurs inconsistantes. Dans ce domaine notre principale originalité provient du fait que nous avons conçu des algorithmes basés sur des domaines qui sont complètement sous-exploités à ce jour dans le cadre de la programmation par contraintes. En effet, dès 1992 il allait de soi que la programmation par contraintes devait utiliser certains algorithmes provenant de la recherche opérationnelle (couplage, flot, arbitrage pour l'ordonnancement disjonctif). En revanche, dans le domaine des contraintes globales, nous avons été l'un des premiers à concevoir des algorithmes de filtrage basés sur des méthodes géométriques [BELDICEANU 00a] ou sur la reconnaissance de langage par des automates [CARLSSON & BELDICEANU 02a], [CARLSSON & BELDICEANU 02b].
– finalement une contribution récente concerne l'identification de familles de contraintes globales et d'algorithmes de filtrage génériques valables pour une famille donnée de contraintes [BELDICEANU & CARLSSON 01a], [BELDICEANU 01b]. Ceci fut motivé par le fait qu'il existe un nombre trop

[22] Lorsqu'un paramètre est une variable cela signifie que l'on pourra poser des contraintes mentionnant la variable en question. Ainsi le fait d'avoir des variables pour la durée et la quantité de ressource consommée par une tâche permet de raisonner en terme de charge de travail. Pour cela on impose une contrainte sur la surface de la tâche en question.

important de contraintes globales et que c'est une tâche démesurée que de faire un algorithme spécifique pour chaque contrainte globale.

Nous avons organisé nos contributions sous les quatre rubriques suivantes :

- **les contraintes globales dans le domaine de l'ordonnancement avec contraintes de capacité.** On considère le cas des ressources renouvelables et celui des ressources non renouvelables. Il s'agit d'une problématique possédant un champ d'application important dans le domaine industriel. On peut penser que c'est le domaine où la programmation par contraintes a le plus réussi en pratique. En dehors de la contrainte *cumulative* [AGGOUN & BELDICEANU 92] qui se retrouve dans quasiment tous les systèmes de programmation par contraintes, une contribution importante concerne l'introduction d'un nouveau modèle de tâches permettant la prise en compte de consommation variable dans le temps [PODER, BELDICEANU & SANLAVILLE 02].
- **les contraintes globales et la géométrie.** Dans ce cadre nous avons conçu des contraintes géométriques [BELDICEANU & CARLSSON 01b], [BELDICEANU, CARLSSON & THIEL 02]. Certaines de ces contraintes, telle que la contrainte *diffn* [BELDICEANU & CONTEJEAN 94], ont été reprises dans un grand nombre de systèmes. Nous avons également montré comment utiliser des techniques traditionnelles de géométrie algorithmique pour traiter certaines contraintes [BELDICEANU, CARLSSON & THIEL 03] se situant à priori en dehors du cadre géométrique.
- **techniques spécifiques pour les algorithmes de filtrage.** Nous avons regroupé dans cette rubrique des techniques originales ne provenant pas de la recherche opérationnelle. Nous considérons successivement des techniques basées sur :
  - le comptage d'événements particuliers [BELDICEANU & CARLSSON 01a],
  - la construction d'automates reconnaissant le langage associé aux solutions acceptées par une contrainte globale [CARLSSON & BELDICEANU 02a], [CARLSSON & BELDICEANU 02b].
- **contraintes avec fonction de coût.** Dans ce domaine nous avons introduit une nouvelle contrainte [BELDICEANU, CARLSSON & THIEL 02] correspondant à un problème d'affectation dont le coût dépend uniquement des valeurs utilisées dans la solution finale.

3.1 Ordonnancement avec contraintes de capacité

Nous indiquons le cheminement ayant conduit à nous intéresser à ce type de problématique. Nous commençons par les problèmes d'ordonnancement [CARLIER & CHRÉTIENNE 88], [PARKER 95] dans lesquels les tâches utilisent des quantités constantes de ressources durant toute leur exécution pour aboutir aux problèmes dans lesquels les consommations des tâches sont variables dans le temps. Ces variations peuvent être discrètes dans le cas de main d'œuvre ou continues dans le cas d'énergie comme l'électricité.

C'est un exercice du ROSEAUX [ROSEAUX 85] sur un problème de chargement-déchargement de bateau qui a, dès 1989, suscité notre intérêt concernant la notion de contrainte cumulative. Plus précisément, on avait un ensemble de tâches soumises, d'une part à des contraintes de précédence, et d'autre part à une contrainte imposant à ne pas dépasser le plafond de main d'œuvre disponible. Chaque tâche nécessitait un nombre donné de personnes pendant toute la durée de son exécution.

Par la suite ce problème est revenu de manière fréquente dans un grand nombre d'applications. Cela a donc motivé le développement de la contrainte *cumulative* de CHIP. Cette contrainte a été pour la première fois présentée en 1992 aux journées francophones de programmation logique à Lille dans [AGGOUN & BELDICEANU 92]. Elle a ensuite été reprise l'année suivante dans l'article [AGGOUN & BELDICEANU 93b] ainsi que par Claude Le Pape dans ILOG Schedule [LE PAPE 94]. Plus tard, la contrainte *cumulative* a également été intégrée sous une forme ou une autre dans la majorité des systèmes de programmation par contraintes sur les domaines finis tels que ECLiPSe [ECLIPSE], IF/Prolog [IF], SICStus [CARLSSON, OTTOSSON & CARLSON 97], OZ [SMOLKA 96] et Claire [CASEAU & LABURTHE 96b].

Noter que, bien souvent, c'est une version restreinte de la contrainte *cumulative* proposée en 1992 qui a été reprise. Cette restriction concerne le fait que certains arguments doivent être a priori fixés. Nous rappelons brièvement la définition de la contrainte *cumulative* de CHIP [AGGOUN & BELDICEANU 92] et nous indiquons certains arguments supplémentaires ayant été introduits par la suite de même que les principales applications de cette contrainte.

Dans un deuxième temps, nous présentons une extension de la contrainte *cumulative* que nous avons développée à SICS [BELDICEANU & CARLSSON 02] dans laquelle nous autorisons les consommations de ressource négatives. Finalement, nous terminons cette partie en présentant brièvement la contrainte *cumulative-trapèze* développée dans le cadre de la thèse

d'Emmanuel Poder [PODER 02] pour des problèmes de consommation ou de production de ressource continues.

3.1.1 La contrainte «*cumulative*»

La contrainte cumulative [AGGOUN & BELDICEANU 92] fut utilisée pour résoudre[23] des problèmes d'ordonnancement dans des domaines tels que :

- la gestion d'atelier [BISDORFF, LAURENT & PICHON 95],
- la maintenance de réseau électrique [CREEMERS, GIRALT, RIERA, FERRARONS, ROCA & CORBELLA 95],
- l'établissement d'emplois du temps [BOIZUMAULT, DELON & PÉRIDY 93], [GOLTZ & MATZKE 99],
- la synthèse de circuits [KUCHCINSKI 98].

La contrainte *cumulative* prend, d'une part un ensemble de tâches dans lequel chaque tâche est définie par son origine, sa durée ainsi que la quantité de ressources consommées, et d'autre part un plafond maximum de ressource à ne pas dépasser. La contrainte *cumulative* impose qu'à chaque instant i, si l'on considère l'ensemble des tâches coupant i, le cumul de leur quantité de ressources consommées n'excède pas le plafond maximum de ressource disponible.

Dans la contrainte *cumulative* originale, l'origine, la durée de même que la quantité de ressources consommées sont des variables sur lesquelles on peut bien entendu imposer des contraintes supplémentaires. Par exemple, lorsque l'on raisonne en terme de charge de travail, on connaît la surface de chaque tâche sans pour autant se fixer initialement la durée ou la quantité de ressources consommées à chaque instant. Finalement dans la contrainte *cumulative* originale le plafond maximum de ressource à ne pas dépasser correspondait également à une variable domaine dont la valeur finale (lorsque toutes les tâches sont complètement fixées) représentait l'altitude du pic de consommation maximum. Cela permettait d'utiliser cette variable hauteur comme un critère à minimiser lorsque l'on se posait la question du niveau minimum de ressource nécessaire pour finir l'ensemble des tâches avant une date donnée.

J'ai introduit par la suite différents paramètres [COSYTEC 02] pour prendre en compte les particularités suivantes :

- il est possible d'indiquer un encadrement de la surface de chaque tâche ainsi qu'un encadrement de la surface totale des tâches,
- il est possible de spécifier une variable de fin générale correspondant au maximum des fins des tâches présentes dans la contrainte *cumulative*,

[23] L'ensemble des références mentionnées dans l'énumération suivante correspond à des applications utilisant la contrainte *cumulative* de CHIP (celle que j'ai réalisée) par des personnes externes à COSYTEC.

– il est possible de préciser la dépendance entre la quantité de ressources consommées par une tâche et sa date de démarrage,
– il est possible d'indiquer un niveau intermédiaire qu'il est absolument nécessaire d'atteindre entre la première et la dernière utilisation de la ressource.

Finalement, nous avons aussi exploité cette contrainte *cumulative* de manière non conventionnelle pour un problème de placement. Elle correspond à une condition nécessaire dans le cas de la contrainte de non-recoupement d'un ensemble de rectangles. Pour ce faire, on oublie la coordonnée sur l'axe des *y* des rectangles et l'on impose une contrainte *cumulative* indiquant que, pour chaque valeur de l'axe des *x*, il ne faut dépasser la hauteur totale disponible de l'espace de placement. Nous avons utilisé avec succès cette condition nécessaire pour le problème du placement sans perte de carrés de tailles distinctes dans un grand carré [SIMONIS & BELDICEANU 99] en se restreignant au cas où la taille des différents carrés est a priori connue[24]. La contrainte *cumulative* a aussi été indispensable pour résoudre le problème de placement parfait de parallélépipèdes décrit dans [BELDICEANU & CONTEJEAN 94].

Pour finir, signalons que la contrainte *cumulative* a été utilisée dans un grand nombre d'applications pratiques [SIMONIS 95], [SIMONIS & CORNELISSENS 95] développées à COSYTEC.

3.1.2 La contrainte «*cumulatives*»

Suite à une étude portant sur un problème de planification de personnel, nous avons proposé en 2002 une généralisation de la contrainte *cumulative* [BELDICEANU & CARLSSON 02] à plusieurs ressources d'où le nouveau nom de contrainte *cumulatives*. Cette contrainte est également utile dans le cadre de gestion de stock intermédiaire dans des systèmes de production.

La contrainte *cumulatives* généralise la contrainte *cumulative* de la manière suivante :
– Tout d'abord, on n'a plus une ressource cumulative mais plusieurs ressources. Cela nécessite l'introduction d'un attribut supplémentaire pour les tâches. Il s'agit d'une variable d'affectation qui indique la ressource à laquelle la tâche est affectée et sur laquelle elle consomme donc effectivement une certaine quantité de ressources pendant son exécution.
– La deuxième extension concerne le fait que l'on autorise également une consommation négative pour une tâche. En clair cela signifie que dans un tel cas une tâche produit de la ressource pendant son exécution.

[24] Mentionnons que la taille des carrés n'est pas fournie dans le problème initialement traité par [DUIJVESTIJN 78]. Ce problème a été repris par Ian Gambini dans sa thèse [GAMBINI 99].

– Enfin, on peut dans cette nouvelle contrainte *cumulative*, soit imposer de ne pas dépasser un plafond maximum de ressource, soit forcer d'atteindre un niveau minimum donné. Cette contrainte ne s'applique bien entendu que pour les instants qui sont chevauchés par au moins une tâche.

Soulignons que si la première extension existait déjà en recherche opérationnelle, la deuxième est complètement nouvelle. Quant à la dernière extension elle existait déjà plus ou moins en programmation par contraintes. D'un point de vue pratique la deuxième extension se justifie par le fait que, bien souvent, on a des problèmes d'ordonnancement dans lesquels il faut placer des tâches de manière à assurer un équilibre entre les tâches qui produisent et les tâches qui consomment la ressource. Nous avons développé trois algorithmes de filtrage pour traiter cette contrainte :

– Le premier algorithme est basé sur une méthode de balayage qui construit de manière incrémentale un profil optimiste de consommation de ressource.
– Le deuxième algorithme élague les attributs d'une tâche en faisant l'hypothèse qu'elle passe sur les différentes ressources où elle peut potentiellement s'exécuter et en n'enlevant effectivement que les valeurs impossibles sur toutes les ressources.
– Le troisième algorithme généralise les méthodes d'intervalles de tâches au cas où l'on a plusieurs ressources et où l'on peut avoir une consommation négative.

Soulignons la simplicité de ces algorithmes qui permettent cependant d'élaguer tous les attributs des tâches (i.e. origine, durée, quantité de ressources, ressource à laquelle est affectée la tâche) aussi bien dans le cas où il ne faut pas dépasser un plafond maximum de ressource que dans le cas où il faut atteindre un seuil minimum de consommation. Une des applications de la contrainte *cumulatives* concerne le cas de ressources non-renouvelables dans lequel il faut éviter d'avoir des stocks négatifs.

3.1.3 La contrainte «*cumulative-trapèze*»

Dans bien des cas pratiques les tâches consomment de la ressource de manière continue ce que l'on ne peut pas modéliser facilement avec la contrainte *cumulative* classique. On peut bien entendu discrétiser une tâche en un ensemble de tâches contiguës dont les origines sont reliées par des contraintes de distances. Malheureusement cela a pour effet de « tuer » la déduction et d'augmenter la taille mémoire nécessaire à la bonne exécution du programme.

C'est pour remédier à cet état de fait que la contrainte *cumulative-trapèze* a été développée. Nous avons introduit un nouveau modèle de tâche dans lequel une tâche est maintenant représentée par une suite de trapèzes contigus. De la même manière que pour la contrainte *cumulative* classique, la contrainte *cumulative-trapèze* impose de ne pas dépasser un plafond maximum de

ressource donné. Le modèle de tâche est le suivant : on associe à chaque tâche les trois variables *origine* de la tâche, *durée totale* de la tâche et *fin* de la tâche ; finalement, à chacun des trapèzes on associe également trois variables représentant la *hauteur gauche* du trapèze, la *hauteur droite* du trapèze et la *longueur de la base* du trapèze. Notons que l'on peut éventuellement contraindre la pente d'un trapèze à être située dans une fourchette donnée. Cela est particulièrement utile pour des problèmes où l'on a des restrictions sur la vitesse de production ou de consommation d'une ressource. Mentionnons que cette contrainte a effectivement été utilisée avec succès dans le cadre du projet européen « GROWTH project G1RD-1999-00034, LISCOS » sur un problème soumis par BASF.

Dans [PODER, BELDICEANU & SANLAVILLE 02] nous avons décrit l'un des algorithmes utilisés dans le cadre de la contrainte *cumulative-trapèze*. Il s'agit du calcul de la partie obligatoire d'une tâche. La notion de partie obligatoire a été, à l'origine, introduite par Lahrichi [LAHRICHI 82] dans sa thèse sur les problèmes cumulatifs. Il s'agit d'un profil de consommation minimale de la tâche quel que soit le placement de son origine dans l'intervalle débutant au début au plus tôt et finissant au début au plus tard de la tâche. Dans le cadre d'une tâche de forme rectangulaire, le calcul de la partie obligatoire revient à faire l'intersection de la tâche au plus tôt et de la tâche au plus tard en prenant sa durée minimale ainsi que sa consommation minimale. Malheureusement ce n'est plus si simple dans le cas d'une tâche formée par une suite de trapèzes. L'article montre comment calculer la partie obligatoire d'une tâche en se basant uniquement sur deux instances réalisables R_1 et R_2 particulières de la tâche dont on fait l'intersection. De plus, lorsque la suite de trapèzes comporte des vallées, (i.e. une diminution de la consommation instantanée suivie par une augmentation) on crée une tâche rabot R à partir des différents fonds de vallées. Finalement dans ce dernier cas on obtient la partie obligatoire en faisant l'intersection entre R_1, R_2 et la tâche rabot R.

D'un point de vue technique notons que l'adaptation des méthodes d'intervalles de tâches [CASEAU & LABURTHE 96b] au cas de la contrainte *cumulative-trapèze* reste un problème à ce jour ouvert. En effet, établir la valeur exacte de l'intersection minimale d'une tâche pas complètement fixée avec un intervalle donné ne semble guère facile.

3.1.4 Conclusion

La contrainte *cumulative* fut la première contrainte globale ayant été utilisée avec succès dans un grand nombre de problèmes pratiques. La contrainte *cumulative* fut étendue par Philippe Baptiste, Claude Le Pape et Wim Nuijten pour prendre en compte la préemption et les tâches élastiques. On peut s'étonner que des contraintes gérant les problèmes de stock ou de production continue n'aient pas été abordées plus tôt par le monde académique, vu leur

grande importance d'un point de vue industriel. D'autres types de contraintes *cumulative* plus générales pourraient voir le jour dans un avenir proche. Enfin, dans le dernier paragraphe, nous avons vu poindre les problèmes géométriques. C'est un point fondamental si l'on veut étendre le champ d'application de la programmation par contraintes. En effet la vie courante n'est pas constituée que de droites et d'angles droits.

3.2 Géométrie algorithmique et contraintes globales

La géométrie algorithmique est à plusieurs titres un domaine important vis-à-vis de la programmation par contraintes :

- tout d'abord, si l'on veut étendre le champ applicatif de la programmation par contraintes, la géométrie algorithmique est un outil de choix pour implémenter les algorithmes de filtrage associés à des contraintes géométriques. La contrainte *cumulative-trapèze* que nous venons de voir illustre bien cette tendance.
- d'autre part, même en se cantonnant à un certain nombre de contraintes non-géométriques, l'adaptation de méthodes de la géométrie algorithmique peut s'avérer extrêmement utile pour dériver des algorithmes efficaces de filtrage. Ces techniques n'avaient pas vraiment été exploitées par le passé. Une idée centrale sous-jacente à ce thème est la suivante : une contrainte peut être vue comme un ensemble de régions interdites. On peut alors interpréter ces régions interdites comme des objets géométriques et raisonner dessus en tirant partie d'éventuelles propriétés de ces objets.

Nous allons détailler les deux points précédents et montrer notre contribution pour chacun d'eux.

3.2.1 Contraintes géométriques

Notre premier intérêt pour les contraintes géométriques date de l'ECRC en 1989 et provient d'une part de la lecture d'un rapport de recherche concernant la partie obligatoire[25] [LAHRICHI & GONDRAN 84], et d'autre part de la lecture du livre sur la géométrie algorithmique de Preparata et Shamos [PREPARATA & SHAMOS 85].

Suite à cela nous avons conçu et développé une contrainte de non-intersection entre des boîtes rectangulaires multidimensionnelles. Cette contrainte dénommée *diffn* [BELDICEANU & CONTEJEAN 94] fut disponible dans CHIP dès 1994. Le choix d'avoir plus de deux dimensions fut motivé

[25] La partie obligatoire d'un objet géométrique correspond à l'intersection de toutes ses instances réalisables.

par le nombre important d'applications[26] que l'on peut modéliser avec une telle abstraction. Par la suite nous avons progressivement enrichi les arguments de cette contrainte afin de prendre en compte des conditions supplémentaires qui sont apparues dans des applications concrètes.

Depuis que nous sommes à SICS nous avons proposé un certain nombre de nouvelles contraintes géométriques dans le catalogue de contraintes globales. Nous avons également généralisé la contrainte *diffn*. Nous allons maintenant revenir de manière plus approfondie sur chacun des points précédents.

3.2.1.1 La contrainte diffn

Dans sa version de base la contrainte *diffn* impose qu'un ensemble de boîtes rectangulaires multidimensionnelles dont les côtés sont parallèles aux axes de l'espace de placement ne se coupent pas deux à deux. D'un point de vue pratique le fait d'avoir un nombre non fixé de dimensions permet d'utiliser la contrainte *diffn* dans des contextes variés. Concrètement les dimensions de l'espace de placement peuvent s'interpréter comme :

- une dimension géographique permettant la localisation d'un objet,
- une dimension temporelle permettant la situation d'un objet dans le temps,
- une dimension d'affectation indiquant à quel composant est affecté un objet.

En combinant les dimensions précédentes nous arrivons ainsi à modéliser les problèmes suivants :

- Un problème dans lequel un ensemble de vecteurs possédant le même nombre de composantes doivent être deux à deux différents.
- Un problème d'ordonnancement de tâches dans lequel on doit d'une part placer dans le temps chaque tâche, et d'autre part assigner chaque tâche à une machine, ceci sans que les tâches affectées à la même machine ne se coupent. On a dans ce cas une dimension temporelle ainsi qu'une dimension d'affectation.
- Un problème de placement de rectangles (les bords de chaque rectangle étant parallèles à l'espace de placement) en imposant la contrainte que les rectangles ne se chevauchent pas deux à deux.
- Un problème de placement de parallélépipèdes similaire au problème précédent.
- Un problème de placement de parallélépipèdes dans des camions qui en plus de l'emplacement dans les camions nécessite une dimension supplémentaire associée à l'ensemble des camions.
- Un problème de placement de parallélépipèdes dans des camions dans lequel on veut en plus modéliser le temps de séjour de chaque

[26] Nous indiquerons ces applications dans le paragraphe concernant la contrainte *diffn*.

parallélépipède dans le camion où il est affecté. La contrainte que l'on assure est que, pour chaque camion et à tout instant, on n'a qu'un seul parallélépipède à un point donné du camion en question. Dans ce problème on a trois dimensions géographiques, une dimension d'affectation et finalement une dimension temporelle.

A côté de cette contrainte de base nous avons, par la suite, ajouté la possibilité de prendre en compte des contraintes telles que :

- le volume minimal ou maximal d'un objet,
- la distance (de Manhattan) minimale ou maximale entre deux objets, le vis-à-vis minimal ou maximal entre deux objets.
- la contrainte d'accessibilité d'un objet par rapport à un ensemble de dimensions et de directions données. L'idée intuitive associée à cette contrainte est que l'on veut garantir un ordre de sortie des objets placés de manière à ce que lorsque l'on translate les objets dans une dimension et direction donnée on ne rencontre pas d'obstacle.

La contrainte *diffn* a été utilisée pour résoudre des problèmes de configuration, de placement, et d'ordonnancement ou d'affectation. Un des problèmes les plus complexes d'un point de vue modélisation résolu avec la contrainte *diffn* et la contrainte *cycle* concerne un problème de chargement-déchargement d'une flotte de véhicules [BOURREAU 99] dans lequel on doit également placer les objets de manière à tenir compte de l'accessibilité : lorsque l'on arrive à un endroit donné pour décharger un colis on désire que ce colis soit placé de telle manière à ce que l'on y ait directement accès. Réciproquement lorsque l'on charge un nouveau colis on ne veut ni déplacer, ni tourner autour des colis déjà placés. Côté placement ce problème est modélisé par une contrainte *diffn* à cinq dimensions : trois géographiques (la situation dans le camion), une d'affectation (le camion auquel le colis est affecté) et une temporelle (le temps de séjour du colis dans le camion).

Terminons en mentionnant que la contrainte *diffn* a été reprise dans un certain nombre de systèmes tels que Bprolog ou IF/PROLOG.

3.2.1.2 La contrainte de non-recoupement entre des rectangles

Dans [BELDICEANU & CARLSSON 01b] nous avons cherché un algorithme de filtrage efficace pour la non-intersection d'un ensemble de rectangles dont les côtés sont parallèles aux axes de l'espace de placement. Bien entendu comme il est d'usage en programmation par contraintes ces rectangles ne sont a priori pas fixés. Plus précisément, nous associons à chaque rectangle R un point origine $R(o)$ correspondant au coin inférieur gauche de R. Ce point possède deux coordonnées $R(o_x)$ et $R(o_y)$ qui sont des variables domaine. L'algorithme que nous avons établi est basé sur l'idée suivante. Pour élaguer

le domaine de la variable $R(o_x)$ d'un rectangle ne devant pas couper un ensemble de rectangles $R_1, R_2, \ldots, R_n$ nous procédons comme suit :

- dans un premier temps, on commence par calculer pour chaque rectangle R_i la portion de l'espace qui est interdite à l'origine de R si l'on ne veut pas que R coupe R_i ; Cette portion est un rectangle qui est éventuellement vide.
- dans un deuxième temps, on utilise une méthode de balayage qui parcourt le domaine de la coordonnée $R(o_x)$ que l'on veut élaguer. Pour chaque position x de la droite de balayage on cherche s'il existe au moins une valeur y appartenant à $R(o_y)$ telle que le point de coordonnées x, y n'appartienne pas à aucune des régions interdites calculées à l'étape précédente. Lorsque l'on ne trouve pas une telle valeur y on peut enlever la valeur x du domaine de la variable $R(o_x)$.

La mise en œuvre efficace de l'idée précédente se fait comme suit :

- A chacun des rectangles interdits pour l'origine de R on associe les deux événements suivants : un événement de début de rectangle interdit et un événement de fin de rectangle interdit. Ces événements sont repérés par leur coordonnée sur l'axe des x et sont triés par ordre croissant de ces coordonnées.
- On associe à la droite de balayage une structure de données permettant de maintenir incrémentalement le nombre de régions interdites coupant un point de coordonnées x, y dans lequel x représente la position courante de la droite de balayage.

Finalement, pour élaguer le domaine de la variable $R(o_y)$ on effectue un balayage sur l'axe des y. On doit également faire le même traitement (i.e. deux balayages) pour élaguer les coordonnées des origines des autres rectangles.

Cependant, dans le contexte de la programmation par contraintes l'algorithme que nous avons établi précédemment doit être réévalué à chaque variation de la valeur minimale ou maximale d'une coordonnée de l'origine d'un rectangle. Nos différentes optimisations de l'algorithme précédent sont motivées par les observations suivantes :

- au fur et à mesure que l'on descend dans l'arbre de recherche, les rectangles sont progressivement fixés.
- en règle générale les domaines des coordonnées des origines des rectangles varient peu entre deux appels successifs à l'algorithme de filtrage que nous venons d'exposer.

Voici les idées intuitives sous tendantes à nos optimisations :

- en sortie de la procédure de filtrage nous mémorisons pour chaque rectangle des points réalisables pour lesquels le rectangle en question ne coupe aucun des autres rectangles. Nous appelons ces points des *témoins*.

Ces témoins sont utilisés comme un filtre peu coûteux pour tester s'il est a priori utile d'exécuter l'algorithme de balayage décrit précédemment lors du prochain appel à l'algorithme de filtrage.

- au fur et à mesure que l'on descend dans l'arbre de recherche les coordonnées de certains rectangles deviennent complètement fixées. On partitionne ces rectangles complètement fixés en deux catégories : d'une part les rectangles qui ne peuvent plus être à la source d'aucun élagage de valeur ; d'autre part les rectangles pouvant causer un élagage. On peut tranquillement écarter la première catégorie de rectangles et bien entendu on n'a plus à élaguer les coordonnées des origines des rectangles de la seconde catégorie.

Finalement, mentionnons un problème ouvert dans le cadre de la contrainte de non-recoupement de rectangles. Il serait utile de trouver une caractérisation précise de l'espace interdit pour l'origine d'un rectangle ceci en tenant compte de l'ensemble des contraintes de non-recoupement. Bien que le calcul effectif de cet espace est de manière quasi-certaine non polynomiale en le nombre des rectangles, cela permettrait malgré tout d'obtenir de façon systématique des règles de propagation utilisables pour un nombre restreint de rectangles.

3.2.1.3 La contrainte de non-recoupement entre des familles de polytopes

Plus récemment dans [BELDICEANU, GUO & THIEL 01] nous avons cherché à généraliser la contrainte *diffn* de manière à prendre en compte des objets dont les côtés ne sont pas parallèles à l'espace de placement. La forme d'un objet O que nous considérons ici est définie par l'enveloppe convexe d'un ensemble fini de points. Un point particulier o est choisi comme l'origine de l'objet O et l'on obtient une famille d'objets $F(O)$ en translatant la forme de O de manière à ce que son origine o appartienne à un domaine donné (également défini par l'enveloppe convexe d'un ensemble fini de points). On associe également au point o ses coordonnées qui sont définies par une suite de variables domaine. Lorsque ces variables sont toutes fixées on obtient le positionnement final de l'objet O. Nous nous sommes intéressés à la contrainte de non-recoupement entre deux familles d'objets $F(O_1)$ et $F(O_2)$ telles que nous venons de les définir. Pour ce faire nous avons commencé dans un premier temps par caractériser la portion de l'espace E vérifiant les propriétés suivantes :

- pour tout point p de E, si l'on fixe l'origine o_1 à p alors O_1 coupe immanquablement toutes les translations de O_2 étant compatibles avec le domaine de son origine o_2,
- pour tout point p n'appartenant pas à E, si l'on fixe l'origine o_1 à p alors il existe au moins une translation de O_2 qui est à la fois compatible avec le domaine de son origine o_2 et qui ne coupe pas O_1.

Dans un deuxième temps nous avons donné deux algorithmes élaguant l'origine d'un objet O_1 de manière à empêcher l'origine de O_1 d'être incluse dans cette région interdite par rapport à un objet O_2 dans les cas particuliers où les objets sont soit des polygones, soit des boîtes rectangulaires multidimensionnelles dont les côtés sont parallèles aux axes de l'espace de placement. Dans le premier cas, nous avons utilisé un algorithme de balayage alors que le deuxième cas évalue simplement certaines conditions. Finalement, nous avons conclu par le fait que ces résultats sont facilement transposables au cas où l'ensemble des points interdits pour l'origine d'une forme fixe O_2 par rapport à une autre forme fixe O_1 est défini par l'enveloppe convexe d'un ensemble fini de points. C'est par exemple le cas de la contrainte de non-inclusion ou de la contrainte de distance minimale de Manhattan entre deux objets. Une question restant en suspend concerne un traitement plus global de la contrainte de non-recoupement similaire à ce que nous avons déjà fait dans le cas des rectangles [BELDICEANU & CARLSSON 01b]. En effet, on voudrait élaguer le domaine de l'origine de l'objet O_1, non pas seulement par rapport au fait qu'il ne doit pas couper O_2, mais également par rapport à l'ensemble des objets que O_1 ne doit pas couper.

3.2.2 Géométrie algorithmique pour le traitement de certaines contraintes

En dehors de leur utilité évidente dans le cadre de contraintes géométriques nous avons montré comment utiliser certaines méthodes provenant de la géométrie algorithmique pour obtenir des algorithmes de filtrage pour des contraintes n'étant a priori pas considérées comme des contraintes géométriques. L'idée centrale sous-jacente est de raisonner globalement sur un ensemble de régions interdites provenant d'une conjonction de plusieurs contraintes. Comme il est généralement coûteux de raisonner avec un nombre important de dimensions on projette les régions interdites sur certains sous-ensembles de variables apparaissant dans les contraintes.

Nous avons adapté la méthode de balayage pour des conjonctions de contraintes ayant certaines structures bien particulières. Nous allons d'abord définir les structures que nous avons considérées puis indiquer les principes généraux utilisés. Jusqu'à maintenant, nous nous sommes penchés sur les trois structures suivantes :

- une conjonction de contraintes dans laquelle chacune des contraintes mentionnent deux variables données [BELDICEANU 00a],
- n conjonctions de contraintes telles que toutes les contraintes de la i-ème conjonction mentionne deux variables X et Y_i ; de plus on a des contraintes mentionnant seulement les variables Y_i [BELDICEANU, CARLSSON & THIEL 03],

– une conjonction de contraintes dans laquelle chacune des contraintes mentionnent deux variables données et pour laquelle on ne désire que satisfaire un nombre de contraintes situé dans une fourchette donnée [BELDICEANU & CARLSSON 01c].

Soulignons que ces structures apparaissent de manière naturelle dans le cadre de divers problèmes de placement. Nous allons d'abord montrer comment représenter les contraintes élémentaires présentes dans les conjonctions de contraintes.

Les différents algorithmes reposent sur la projection par rapport à deux variables données X et Y d'une contrainte pouvant éventuellement comporter d'autres variables dénotées par l'ensemble V. Les points de cette projection par rapport à deux variables particulières se répartissent dans les trois ensembles suivants :

– l'ensemble des points où la contrainte est toujours satisfaite quelle que soit la valeur prise par les variables de l'ensemble V (régions sûres),
– l'ensemble des points où la contrainte n'est jamais satisfaite quelle que soit la valeur prise par les variables de l'ensemble V (régions interdites),
– l'ensemble des points restants.

La figure 1 illustre cette décomposition pour cinq contraintes. Elle montre en grisé les régions interdites et en hachuré les régions sûres par rapport à deux variables données et leur domaine[27] respectif. La première contrainte (A) impose que X, Y, $4-Y$ et R prennent des valeurs distinctes[28] tandis les autres contraintes correspondent à des contraintes arithmétiques (B, C, E) et à une contrainte disjonctive (D).

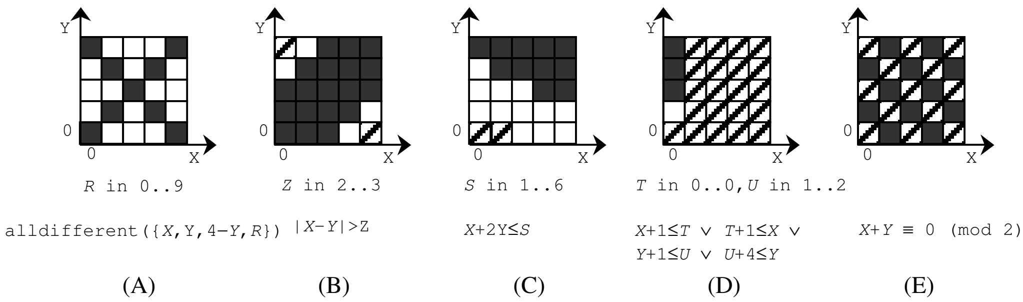


Fig. 1. Exemples de régions interdites et de régions sûres pour $0 \leq X \leq 4$, $0 \leq Y \leq 4$.

[27] La déclaration *Var* in *min..max* indique que l'on a une variable domaine *Var* dont le domaine initial est constitué par toute les valeurs comprises entre *min* et *max*.

[28] Nous n'avons pas éliminé la valeur 2 du domaine de la variable *Y*, car nous avons supposé que la contrainte *alldifferent* ne prend pas en compte le fait qu'une même variable apparaît dans plus d'un terme.

Dans ce cadre, définir une nouvelle contrainte revient donc à fournir un ensemble de primitives renvoyant les points où la contrainte est toujours satisfaite ainsi que les points où la contrainte n'est jamais satisfaite. Ces points sont regroupés dans des rectangles qui décrivent une portion de l'espace dans laquelle tous les points ont la même nature.

Par la suite, pour détecter les valeurs impossibles du domaine d'une variable en fonction d'une conjonction de contraintes où elle apparaît, nous utilisons une méthode de balayage qui mémorise dans une structure de données appropriée les rectangles coupant la droite de balayage. C'est uniquement la structure de données qui change en fonction des trois types de conjonctions de contraintes que nous avons considérées.

Une voie prometteuse concerne la généralisation des méthodes précédentes au cas où les contraintes partagent plus de deux variables et au cas général où l'on n'a pas de structure particulière sur la répartition des variables dans les contraintes. Dans ce cas il faut pouvoir régler les questions suivantes :

- être capable de calculer les points interdits et autorisés pour une contrainte : cela reste envisageable pour un grand nombre de contraintes courantes,
- être capable de généraliser les structures de données au cas multidimensionnel : cela ne semble envisageable que dans le cadre d'un nombre réduit de dimensions de l'ordre de trois.

Ce dernier problème est apparenté[29] au problème de mesure de Klee[30] [OVERMARS & YAP 91] dont on peut avoir un aperçu à la page (`http://compgeom.cs.uiuc.edu/~jeffe/open/klee.html`) et pour lequel il n'existe malheureusement pas de solution efficace.

3.2.3 Synthèse et perspectives

Nous avons montré comment la géométrie algorithmique intervient à différents degrés dans le cadre de la programmation par contraintes :

- d'une part en offrant des contraintes géométriques, ce qui permet d'élargir le champ d'application de la programmation par contraintes,
- d'autre part en réadaptant des méthodes bien connues de la géométrie algorithmique au traitement de certains types de conjonctions de contraintes.

Nous n'avons fait qu'effleurer ce domaine, et l'avenir devrait être particulièrement riche pour de nouvelles utilisations de la géométrie

---

[29] En fait cela permet de tester directement l'existence d'un point réalisable (i.e. point n'appartenant à aucune région interdite).

[30] Le problème de mesure de Klee peut être décrit comme suit : étant donné un ensemble de boîtes rectangulaires multidimensionnelles dont les côtés sont parallèles aux axes de l'espace de placement, existe-il un algorithme efficace pour calculer le volume de l'union de ces boîtes?

algorithmique dans le domaine des contraintes. Soulignons que ce n'est malheureusement pas un sujet facile. En effet, on ne peut souvent pas retransposer directement les algorithmes comme c'est parfois le cas en ordonnancement.

D'autre part, on est souvent confronté au problème bien connu de la robustesse des algorithmes géométriques due à la précision de calcul [PION 99]. Dans le cadre de la programmation par contraintes les problèmes de robustesse sont cruciaux car les algorithmes de filtrage sont sollicités un grand nombre de fois lorsque l'on développe un arbre de recherche. Des erreurs de calculs peuvent entraîner des arrêts inopinés du programme, la production de solutions incorrectes ou la disparition de solutions.

3.3 Techniques spécifiques pour les algorithmes de filtrage

Nous venons de voir comment la géométrie algorithmique peut être utilisée dans le cadre de la programmation par contraintes. Nous passons maintenant à d'autres domaines que nous avons commencés à explorer et qui ne sont, à ce jour, presque pas exploités dans le cadre de l'établissement d'algorithmes de filtrage efficaces. Dans un premier temps nous présenterons des algorithmes de filtrage basés sur le dénombrement. Finalement, dans un second temps, nous indiquerons une nouvelle méthode pour dériver de manière systématique des algorithmes de filtrage basés sur la construction d'automates reconnaissant le langage associé aux solutions d'une contrainte globale.

3.3.1 Algorithmes de filtrage basés sur le dénombrement

Un certain nombre d'algorithmes de filtrage reposent sur le comptage d'événements particuliers tels par exemple le nombre de fois qu'une valeur est retirée du domaine d'une variable sous certaines conditions. Ces algorithmes peuvent être vus comme une extension du principe de *construction disjonctive* [VAN HENTENRYCK, SARASWAT & DEVILLE 95] dans lequel on impose successivement deux contraintes pour finalement ne retirer les valeurs qui ont vraiment été enlevées dans les deux termes de l'alternative. Nous avons spécialisé cette technique aux cas particuliers suivants :

– Dans [BELDICEANU & CARLSSON 01a] nous revisitons l'opérateur de cardinalité[31] [VAN HENTENRYCK & DEVILLE 91] et spécialisons l'opérateur de cardinalité au cas important où les contraintes se chevauchent de manière régulière : par rapport à une séquence donnée de variables on considère une contrainte sur toutes les sous-séquences formées par n variables consécutives. Nous présentons de nouveaux algorithmes de

[31] L'opérateur de cardinalité impose que parmi un ensemble donné de contraintes, un certain nombre de ces contraintes soient satisfaites.

filtrage basés sur le fait d'imposer certaines conjonctions de contraintes et sur des arguments portant sur le comptage du nombre de fois qu'une valeur est supprimée d'un domaine donné.

- Dans [BELDICEANU & CARLSSON 01c] nous considérons un autre cas particulier de l'opérateur de cardinalité [VAN HENTENRYCK & DEVILLE 91] dans lequel on demande que, parmi n contraintes données ayant au moins deux variables en commun, on ait n_1 ou n_2 ou …ou n_k contraintes satisfaites. Chaque contrainte est définie par ses régions interdites et ses régions sûres par rapport aux deux variables communes. On utilise une méthode de balayage pour propager par rapport au nombre de contraintes devant être satisfaites (bien que l'on ne connaisse pas exactement ce nombre). Pour chaque point de la droite de balayage on compte le nombre de régions interdites et de régions obligatoires la contenant.

Les deux méthodes de comptage précédentes présentent l'avantage d'être assez génériques car elles ne dépendent pas de la spécificité des contraintes rencontrées mais de la structure d'une conjonction de contraintes. Dans l'avenir, nous espérons adapter ces techniques à d'autres structures sur lesquelles nous avons des résultats préliminaires.

3.3.2 Algorithmes de filtrage basés sur des automates

Très récemment, à la fin de l'année 2002, nous nous sommes intéressés aux contraintes permettant d'éliminer de manière active des solutions symétriques. Leur champ d'application concerne les problèmes dans lesquels on a des matrices de variables de décision sur lesquelles on pose des contraintes d'ordre lexicographique entre les colonnes ou les lignes. Notre intérêt pour cette question a été éveillé par le nombre important d'articles autour de ce thème à la dernière conférence sur les contraintes.

Suite à cela, nous avons revisité la contrainte d'ordre lexicographique entre deux vecteurs de variables domaine et fourni un nouvel algorithme pour traiter globalement le problème d'un ensemble de contraintes d'ordre lexicographique. Ce deuxième problème correspondait à une question ouverte. Dans les deux cas nous avons dérivé de manière systématique l'algorithme de filtrage à partir d'un automate déterministe à états finis reconnaissant les solutions associées à la contrainte. L'algorithme de filtrage n'étant qu'une retranscription de l'automate, la validité de l'algorithme provient de la preuve que l'automate reconnaît seulement le langage associé aux solutions de la contrainte considérée. Plus précisément notre contribution dans ces deux problèmes à été la suivante :

- Le premier article [CARLSSON & BELDICEANU 02a] revisite la contrainte d'ordre lexicographique entre deux vecteurs de variables domaine. Un premier algorithme avait été proposé par Frisch et al. [FRISCH, HNICH,

KIZILTAN, MIGUEL & WALSH 02] mais il était breveté et donc inutilisable en l'état dans SICStus Prolog. Nous avons développé un nouvel algorithme dont la complexité dans le pire des cas est similaire à celle de l'algorithme précédent. Il permet cependant de détecter certaines configurations, dans lesquelles la contrainte est forcément satisfaite, qui n'étaient pas décelés par l'algorithme précédent. L'originalité principale de notre algorithme provient du fait qu'il est dérivé de manière systématique d'un automate déterministe à états finis opérant sur une chaîne capturant la relation entre chaque paire de variables des deux vecteurs considérés dans la contrainte d'ordre lexicographique.

– Le deuxième article [CARLSSON & BELDICEANU 02b] résout la question posée dans la conclusion du même article mentionné précédemment [FRISCH, HNICH, KIZILTAN, MIGUEL & WALSH 02] sur la contrainte d'ordre lexicographique. La question posée était la suivante : est-il possible d'avoir un algorithme polynomial complet (c'est-à-dire détectant absolument toutes les valeurs interdites) pour une chaîne de contraintes lexicographiques entre une suite de vecteurs de variables domaine. Cette question est importante dans le cadre de problèmes de décisions où l'on a une matrice de variables domaine sur laquelle on peut imposer un ordre lexicographique entre les colonnes adjacentes de la matrice, ceci dans le but d'éliminer certaines symétries. Là encore l'algorithme de filtrage que nous avons proposé est basé sur la même idée que l'algorithme précédent.

Finalement, l'utilisation d'automates déterministes à états finis reconnaissant le langage associé aux solutions d'une contrainte devrait également s'avérer utile dans le cadre de contraintes portant sur des mots [CROCHEMORE, HANCART, LECROQ 01]. Nous sommes en train de réutiliser cette technique pour un problème ouvert toujours dans le cadre de contraintes permettant d'éliminer des solutions symétriques.

3.4 Contraintes d'optimisation

La programmation par contraintes a bien souvent été dénoncée par la communauté recherche opérationnelle pour sa faiblesse dans le domaine de l'optimisation. C'est pour cette raison que depuis quelques années plusieurs chercheurs tels que Philippe Baptiste, Yves Caseau, Torsten Fahle, Claude Le Pape, François Laburthe ou Michela Milano ont commencé par introduire des contraintes «d'optimisation» qui, la plupart du temps, réadaptent certains algorithmes connus au contexte de la programmation par contraintes (incrémentalité, gestion du retour arrière). Cela a par exemple été fait dans le domaine de l'ordonnancement [BAPTISTE, LE PAPE & PERIDY 98][32], dans le

[32] Dans ce cas précis il s'agit d'un nouvel algorithme.

cas de problèmes d'affectation [CASEAU & LABURTHE 97a] ainsi que pour le problème de la détermination de clique maximale [FAHLE 02]. Le principe général de ces contraintes d'optimisation est le suivant : pour une fonction de coût particulière on crée une contrainte d'optimisation dont les arguments sont, d'une part les variables de décision intervenant dans la fonction de coût, et d'autre part la variable de coût proprement dite. A cette contrainte d'optimisation on associe généralement les deux types d'algorithmes suivants :

- un premier algorithme évalue, à partir de l'état courant des variables de décision, une borne inférieure de la variable de coût,
- un deuxième algorithme élague les variables de décision de manière à ne pas dépasser la valeur maximale de la variable de coût.

Notre contribution [BELDICEANU, CARLSSON & THIEL 02] dans ce domaine est la suivante. Nous nous sommes intéressés à une fonction de coût, qui à notre connaissance, n'a pas été traitée en recherche opérationnelle, mais qui cependant a de nombreuses applications pratiques. Il s'agit en l'occurrence de la *somme des poids associés aux valeurs distinctes* prises par un ensemble de variables de décision. Nous avons donc introduit la contrainte « somme des poids associés aux valeurs distinctes » et fourni des algorithmes de filtrage pour les différents aspects de cette contrainte. Cette contrainte généralise la contrainte *alldifferent* et la contrainte «nombre de valeurs distinctes». Un premier aspect de la contrainte est relié au problème de domination dans un graphe. Pour cet aspect nous avons développé des algorithmes qui sont complets (c'est-à-dire qui produisent un élagage complet) lorsque chaque variable de la contrainte n'est constituée que d'un seul intervalle de valeurs consécutives[33]. Pour le deuxième aspect qui est connecté au problème de couplage nous avons proposé un algorithme complet. Finalement, nous avons introduit plusieurs règles de déductions prenant simultanément en compte les deux aspects de la contrainte. L'une de ces règles résout partiellement une question posée dans la conclusion d'un article de Régin [RÉGIN 99b] portant sur la contrainte «*global cardinality*».

Concluons ce paragraphe sur les contraintes d'optimisation par l'observation suivante. Si l'on regarde les caractéristiques associées au graphes introduites dans le chapitre suivant (nombre de composantes connexes, nombre de sources, …) on s'aperçoit bien vite que pour chacune de ces caractéristiques nous avons une contrainte d'optimisation. Cela indique qu'il reste donc un travail certain à accomplir dans ce domaine.

---

[33] Lorsque cette hypothèse n'est pas vérifiée ces algorithmes restent valables tout en ne garantissant plus un élagage complet des domaines des variables.

3.5 Conclusion

Nous venons de présenter dans ce chapitre nos contributions concernant l'identification de nouvelles contraintes globales et l'établissement d'algorithmes de filtrage efficaces. Soulignons que, dans la majorité de nos articles décrivant des algorithmes, nous nous sommes attaché à analyser la complexité dans le pire des cas. Mentionnons que, dans ce domaine, nous avons été l'un des premiers à considérer la complexité des opérations d'accès aux variables domaine[34]. Nous avons également été les premiers à concevoir des algorithmes de filtrage tirant explicitement partie d'une représentation donnée des variables domaine [BELDICEANU, CARLSSON & THIEL 02] afin d'avoir une meilleure complexité. En particulier il semble maintenant clair que la représentation des domaines par une liste d'intervalles s'avère être un compromis judicieux pour les algorithmes de filtrage utilisant des algorithmes de graphe (i.e. couplage, flot, calcul de composantes fortement connexes, parcours en profondeur d'abord) dans lesquels on doit parcourir les sommets adjacents à un sommet donné. En effet, cette représentation permet de conserver les complexités des algorithmes précédent, ce qui n'est guère le cas lorsque l'on utilise une représentation par tableau de bits.

Afin de minimiser les problèmes de consommation mémoire, nous nous sommes également attaché à concevoir des algorithmes de filtrage dans lesquels les graphes ne sont pas représentés de façon explicite.

De manière générale, soulignons la richesse des techniques utilisées s'appuyant sur une grande variété de domaines tels que les graphes, la géométrie ou les automates.

[34] Traditionnellement, la plupart des chercheurs font implicitement l'hypothèse simplificatrice suivante : la complexité dans le pire des cas des primitives d'accès aux variables domaine est $O(1)$. Malheureusement, cela est assez grossier en pratique.

4 Contributions sur les aspects déclaratifs

Ce chapitre dégage nos contributions dans le domaine des contraintes globales en se focalisant sur les aspects déclaratifs. C'est à nos yeux un point fondamental largement ignoré par l'immense majorité des chercheurs[35] travaillant dans le domaine des contraintes globales. Nous expliquons en quoi consistent ces aspects déclaratifs et pourquoi ils sont importants pour l'avenir du développement des contraintes.

Le travail sur les aspects déclaratifs consiste à rendre explicites des informations qui, à l'heure actuelle, sont présentes de manière implicite dans les manuels de systèmes de programmation par contraintes ou existent en filigrane dans les algorithmes de filtrage associés aux différentes contraintes. D'une part en se penchant sur les manuels de programmation par contraintes on constate que les contraintes globales sont définies de manière informelle en s'appuyant sur le langage naturel. D'autre part, en se penchant sur le code des algorithmes de filtrage de ces contraintes on constate bien souvent que l'on a perdu des informations cruciales sous-tendant la création de ces algorithmes. Il peut s'agir d'informations concernant le pourquoi du retrait de certaines valeurs du domaine des variables ou d'informations concernant les hypothèses implicites que l'on est amené à faire localement dans un algorithme de filtrage[36].

Le fait que les informations que nous avons évoquées précédemment ne soient pas explicitement disponibles présente les inconvénients majeurs suivants. Chaque contrainte globale est considérée comme un cas particulier vis-à-vis des différents aspects dans lesquels elle intervient :

- l'analyse et le contrôle de ses paramètres lors de sa mise en place,
- l'algorithme de filtrage associé à la contrainte,
- la production d'explications au pourquoi du retrait de valeurs des domaines,
- la visualisation de l'état de contrainte.

Le fait de considérer chaque contrainte globale comme un cas particulier est d'autant plus néfaste qu'il existe un nombre important de contraintes globales. Cela est encore aggravé par l'introduction récente de contraintes globales portant sur des variables de type ensemble[37] ou de type «multiset».

---

[35] En effet, mis à part les travaux de Narendra Jussien [JUSSIEN 01] dans le domaine des explications, nous ne voyons à l'heure actuelle pas vraiment d'autre chercheur travaillant sur les aspects déclaratifs.

[36] Cet aspect sera détaillé dans le paragraphe concernant la communication entre contraintes.

[37] C'est par exemple le cas dans ILOG Configurator où il existe une contrainte *alldifferent* entre des variables de type ensemble.

C'est pourquoi nous considérons que le principal enjeu de la maîtrise des aspects déclaratifs concerne l'obtention, pour les contraintes globales, de code prenant en compte les différents aspects que nous avons mentionnés précédemment (contrôles des paramètres, algorithme de filtrage, production d'explications, visualisation). Ce code serait, soit synthétisé à partir de la description des contraintes, soit obtenu par composition de briques élémentaires.

Nous montrons maintenant nos différents apports concernant les aspects déclaratifs des contraintes globales dans les trois domaines suivants :

- la description explicite du sens des contraintes globales en terme de caractéristiques d'un graphe ; elle nous a permis de représenter plus d'une centaine de contraintes globales,
- les problèmes de communication transparente entre contraintes ; cela permet, dans un algorithme de filtrage, de tenir compte des contraintes extérieures sans pour autant les connaître explicitement,
- le problème d'élaboration d'outils de mise au point pour la programmation par contraintes.

4.1 Classification des contraintes globales

L'objectif de cette section est d'abord de montrer le contexte qui nous a amené à introduire une classification des contraintes globales. Nous donnons un aperçu du principe de classification utilisé et nous terminons en indiquant les perspectives ouvertes par une telle classification.

4.1.1 Évolution de la perception des contraintes globales

Dès leur apparition en 1989, les contraintes globales ont été vivement critiquées par les tenants de l'intelligence artificielle et de la programmation logique. On leur a en effet beaucoup reproché d'être des constructions ad hoc et l'on a ironisé sur le fait qu'il fallait redévelopper une nouvelle contrainte pour chaque nouveau problème que l'on voulait traiter. L'approche contrainte globale a, en revanche, intéressé un certain nombre de personnes provenant de la recherche opérationnelle, car dans ce milieu il allait de soi que chaque problème nécessite un algorithme particulier. Cela a permis d'initier une collaboration fructueuse entre la programmation par contraintes et la recherche opérationnelle surtout dans le domaine de l'ordonnancement [BAPTISTE, LE PAPE & NUIJTEN 01]. Par la suite, après les succès industriels de la programmation par contraintes dans le domaine de l'ordonnancement vers les années 1993-1994, les chercheurs du domaine de l'intelligence artificielle et de la programmation logique ont accepté avec résignation le fait que les contraintes globales puissent bien être utiles dans la pratique.

Pour ma part, bien qu'ayant introduit un nombre important de contraintes globales spécialisées, j'ai trouvé qu'il y avait une part de vérité dans les critiques formulées par les tenants de l'intelligence artificielle et de la programmation logique : sous la pression des applications j'avais été conduit à développer un certain nombre de contraintes ad hoc et je me suis progressivement rendu compte que j'avais redéveloppé assez souvent des variantes de contraintes déjà existantes. Malheureusement je voyais difficilement comment éviter ces contraintes ad hoc car les problèmes réels recélent une variété insoupçonnée de contraintes qui ne sont pas toujours facilement modélisables avec un jeu prédéfini de contraintes. C'est une réalité hélas trop souvent ignorée des milieux académiques mais bien vite constatée par les ingénieurs utilisant un outil de programmation par contraintes pour modéliser des problèmes concrets dans toute leur complexité opérationnelle.

A mon arrivée à SICS j'ai donc essayé de trouver un moyen de décrire explicitement les contraintes globales. Au début cela semblait un peu utopique mais j'ai progressivement réussi à trouver un principe permettant de décrire explicitement le sens des principales contraintes que j'avais conçues par le passé (i.e. *cumulative*, *cycle*, *diffn*, *among*, …). Avant d'indiquer les principes de cette description, je vais d'abord replacer cette démarche dans une perspective plus large.

4.1.2 Motivations et fondements d'une classification

Jusqu'à présent, les problèmes combinatoires ont toujours été décrits au moyen du langage naturel[38], et bien que des algorithmes efficaces aient été développés pour beaucoup de problèmes, ces algorithmes sont trop spécialisés pour être utilisés comme des composants logiciels. Les traités les plus connus mettant en évidence de nombreux problèmes, particulièrement du point de vue de la complexité, sont entre autres :

- « Computers and Intractability: A Guide to the Theory of NP-completeness » de M. R. Garey et D. S. Johnson en 1979,
- « Graph Coloring Problems » de T. R. Jensen et B. Toft en 1995,
- « Complexity and Approximations » de G. Ausiello et al. en 1999.

D'un côté ces ouvrages ont été incontestablement un facteur d'émulation dans la recherche d'algorithmes efficaces pour un nombre important de problèmes. Mais, d'un autre côté, ils ont distillé de manière imperceptible les inconvénients majeurs suivants :

[38] Il y a bien sûr eu l'établissement de notations dans certains domaines spécifiques tels que l'ordonnancement [GRAHAM, LAWLER, LENSTRA & RINNOOY KAN 79]. Cependant, ces notations n'ont pas été introduites dans un souci d'aboutir à une description utilisable par un programme informatique.

- Du point de vue du génie logiciel un problème est *typiquement plus grand* qu'un composant. Cela a conduit à des algorithmes efficaces mais monolithiques ne pouvant pas être automatiquement réutilisés dès lors que le problème initial était légèrement modifié. Cet état de fait est particulièrement néfaste de nos jours où les spécifications d'un problème évoluent rapidement dans le temps et où l'industrie informatique veut mettre en avant le concept de *composant réutilisable*.
- Ces recueils de problèmes ont implicitement supporté le message que l'efficacité ne pouvait être obtenue que si l'on considère chaque problème séparément. C'est cependant une prise de position étrange d'un point de vue scientifique. En effet le but ultime de la plupart des domaines scientifiques est de découvrir les concepts de base cachés derrière des phénomènes apparemment disparates et d'établir un lien entre les propriétés de ces concepts et de ces phénomènes.

Finalement, comme nous l'avons auparavant souligné dans le paragraphe précédent, les contraintes globales étaient dans une situation similaire à celle que nous venons de décrire. Face à ces observations nous nous sommes mis en quête d'un moyen de décrire explicitement le sens[39] des contraintes globales à partir d'un jeu limité de concepts de base. Le choix d'une description basée sur les graphes[40] a été influencé par les observations suivantes :

- le concept de graphe prend racine dans le domaine des récréations mathématiques (voir par exemple Euler, Dudeney, Lucas et Kirkman) qui est en quelque sorte l'ancêtre des problèmes combinatoires actuels ; dans cette perspective une description basée sur les graphes prend tout son sens.
- dans l'un des premiers livres introduisant la théorie des graphes [BERGE 58] Claude Berge introduit la théorie des graphes comme un moyen de rassembler des problèmes et des résultats apparemment disparates ; c'est également le problème auquel nous étions confrontés dans le cadre des contraintes globales.
- finalement, les caractéristiques associées aux graphes sont à la fois concrètes et concises. De plus, elles sont bien souvent reliées à des considérations pratiques.

Nous allons maintenant donner les idées de base que nous avons utilisées pour classifier les contraintes globales. Elles sont décrites dans le rapport de recherche [BELDICEANU 00b] et résumées dans l'article [BELDICEANU 00c].

[39] Dans le cadre des contraintes, c'est également une précondition au développement de l'aspect métaconnaissances proné dans [PITRAT 90].

[40] Par exemple la contrainte sur le *nombre de valeurs distinctes prises par un ensemble de variables* peut être vue comme une contrainte sur le nombre de composantes fortement connexes dans un graphe particulier construit à partir des variables de l'ensemble considéré.

4.1.3 Principe de description des contraintes globales

Nous présentons sous une forme simplifiée le principe de description des contraintes globales. Pour de plus amples détails le lecteur est invité à ce référer au rapport [BELDICEANU 00b]. Une contrainte globale est représentée par un graphe orienté dans lequel chaque sommet correspond à une variable et chaque arc à une contrainte binaire entre les variables associées aux deux extrémités de l'arc en question. Maintenant le fait nouveau par rapport à ce qui se fait habituellement dans les réseaux de contraintes [DECHTER & PEARL 87], est que l'on ne demande plus forcément que toutes les contraintes binaires soient satisfaites. Bien au contraire on considère le graphe précédent dans lequel on retire les arcs associés à toute contrainte binaire non vérifiée et l'on impose le fait que ce graphe possède certaines caractéristiques. Par exemple, on désire que le nombre de composantes fortement connexes de ce graphe soit compris dans une fourchette donnée. Nous avons ainsi introduit plusieurs caractéristiques communes telles que, par exemple, le nombre de composantes connexes, le nombre de composantes fortement connexes, le nombre d'arcs, le nombre de sommets sources ou le nombre de sommets puits.

Il nous reste à indiquer comment générer le graphe initial à partir duquel on cherche à imposer une certaine caractéristique. En effet, une contrainte possède un ou plusieurs arguments qui, la plupart du temps, correspondent à une variable domaine ou à une collection de variables domaine. Il faut donc décrire le processus permettant de passer des arguments d'une contrainte au graphe que nous avons introduit précédemment. Pour ce faire nous décrivons comment générer à partir des arguments de la contrainte les sommets ainsi que les arcs du graphe. A cette fin nous avons créé un jeu prédéfini de générateurs de nœuds et d'arcs. Nous allons donner un exemple concret de description de contrainte globale afin d'illustrer la démarche précédente.

Considérons la contrainte nvalue(N,[$V_1,\ldots,V_m$]) dans laquelle N et [$V_1,\ldots,V_m$] correspondent respectivement à une variable domaine ainsi qu'à une collection de variables domaine. Cette contrainte est satisfaite lorsque N est égal au nombre de valeurs distinctes prises par les variables $V_1,\ldots,V_m$. Nous montrons d'abord quel est le graphe associé à cette contrainte. Puis nous décrirons la contrainte associée à chaque arc du graphe précédent. Finalement, nous terminerons par la caractéristique que nous voulons imposer sur le graphe généré.

A chaque variable de la collection de variables [$V_1,\ldots,V_m$] correspond un sommet du graphe contenant la variable en question. Puis nous générons un arc entre chaque sommet du graphe (y compris entre un sommet et lui-même) et nous associons la contrainte binaire d'égalité à chaque arc. Finalement nous imposons à ce que la variable N contenue dans le premier argument de la contrainte *nvalue* soit égale au nombre de composantes fortement connexes du

graphe que nous venons de générer dans lequel on écartera les arcs associés à des contraintes n'étant pas vérifiées.

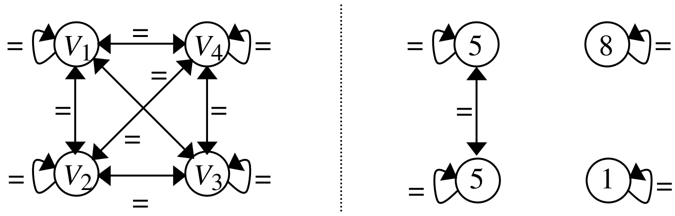


Fig. 2. Graphe initial associé à la contrainte nvalue(N,[V_1,V_2,V_3,V_4]) et graphe final associé à l'instanciation nvalue(3,[5,5,1,8]).

La figure précédente montre le graphe initialement généré pour la contrainte nvalue(N, [V_1,V_2,V_3,V_4]) ainsi que le graphe associé à l'instanciation nvalue(3, [5, 5, 1, 8]). Pour le graphe initial nous indiquons dans chaque sommet la variable associée tandis que pour le graphe final nous indiquons la valeur prise par la variable. Dans le graphe final nous avons enlevé tous les arcs correspondants aux égalités non vérifiées. La contrainte nvalue(3, [5, 5, 1, 8]) est satisfaite car le graphe final possède bien trois composantes fortement connexes ce qui, dans le contexte de la définition associée à la contrainte *nvalue*, se réinterprète comme le fait que N est égal au nombre de valeurs distinctes prises par les variables V_1,V_2,V_3 et V_4.

Nous venons d'illustrer sur un exemple concret de contrainte globale comment passer d'une description en langage naturel du sens de la contrainte vers une description pouvant directement être interprétée par un programme informatique. L'article [BELDICEANU 00c] montre comment procéder pour décrire explicitement la plupart des contraintes globales.

4.1.4 Bilan et perspectives

Le premier catalogue de contraintes globales introduit en janvier 2000 contenait une soixantaine de contraintes globales. Nous avons tenu à jour ce catalogue en y intégrant des contraintes qui manquaient telle que la contrainte *element* [VAN HENTENRYCK & CARILLON 88] ainsi que de nouvelles contraintes apparues récemment dans la littérature [PETIT, RÉGIN & BESSIÈRE 01], [PESANT 01] ou des contraintes que nous avons créées. Nous avons également légèrement modifié la manière de décrire les contraintes afin de prendre en compte certains aspects que nous avions initialement ignorés. A ce jour (janvier 2003) la version courante du catalogue comporte environ 140 contraintes globales, toutes décrites en terme de caractéristiques de graphe.

Enfin soulignons qu'un programme testant si une contrainte, dont tous les arguments sont fixés, a été réalisé pour un sous-ensemble du catalogue par David Hanak de l'université de Budapest [HANAK 02]. Il reste cependant à montrer qu'il est également possible de faire la même chose dans le cas des algorithmes de filtrage. En pratique cela signifie que nous voudrions être capable de synthétiser un algorithme de filtrage pour n'importe quelle contrainte que nous pouvons décrire. C'est un enjeu important car cela permettrait de produire systématiquement un grand nombre d'algorithmes de filtrage, ce qu'il est impossible de faire à l'heure actuelle.

Le lien entre les propriétés des contraintes élémentaires associées aux arcs d'un graphe et les algorithmes de filtrage reste à établir[41]. Finalement, mentionnons que cette classification pourrait avoir des retombées par delà le domaine des contraintes. En effet nous pensons qu'elle pourrait être utile dans le cadre de la recherche locale en proposant de remplacer les mouvements orientés problèmes par des mouvements orientés composants. On peut noter quelques essais dans cette direction de recherche tels que [NAREYEK 01] et [MICHEL & VAN HENTENRYCK 02].

4.2 Communication entre contraintes

Une vision académique traditionnelle de l'informatique concernant la résolution efficace de problèmes préconise la démarche suivante. Dans un premier temps on analyse les propriétés du problème pour finalement aboutir à un algorithme. Le point faible de cette approche est qu'elle nécessite un éternel recommencement à chaque problème. C'est pour essayer de résoudre cet état de fait que l'on a par la suite développé des bibliothèques d'algorithmes [MEHLHORN & NÄHER 99] dans un domaine particulier afin de pouvoir réutiliser des composants éprouvés. Cependant, dans le cadre des contraintes, l'approche courante dans le domaine des composants ne prend pas complètement en compte les aspects suivants :

- la description *explicite* des fonctionnalités offertes par chaque composant,
- la communication entre composants.

Nous avons déjà évoqué le premier point dans la section portant sur la classification des contraintes globales. Nous allons ici nous attacher au problème de la communication entre les composants.

[41] Nous sommes convaincus de l'existence de tels liens mais nous n'avons malheureusement pour l'instant que quelques résultats incomplets.

4.2.1 Problématique

Même si l'on fournit des algorithmes efficaces pour chaque contrainte cela ne permet guère de garantir un comportement global cohérent d'un ensemble de contraintes. En effet, dans la pratique industrielle les personnes un peu avisées utilisant un système de programmation par contraintes sont trop souvent surprises par des manques flagrants de déductions relevant du bon sens commun. C'est d'autant plus frustrant qu'il s'agit de situations ne requérant aucun raisonnement ou algorithme sophistiqué.

4.2.2 Situation actuelle

L'un des principaux apports des systèmes de programmation par contraintes est le remplacement de l'exécution séquentielle d'un algorithme par l'exécution en tant que coroutines d'un ensemble d'algorithmes ne communiquant que par des événements concernant la modification du domaine des variables. Ce mécanisme de coroutine peut être soit directement simulé dans des langages procéduraux classiques, soit obtenu par l'introduction de primitives de contrôles dans des langages plus évolués tels que Prolog. Cette communication par l'intermédiaire des domaines des variables a bien souvent été mise en exergue comme le point fort de l'approche contrainte car elle permet d'introduire de nouvelles contraintes sans se soucier des contraintes préexistantes. Malheureusement, c'est en même temps le talon d'Achille des systèmes actuels car elle repose sur l'hypothèse simplificatrice suivante : chaque algorithme de filtrage suppose implicitement que chaque variable peut effectivement prendre n'importe quelle valeur encore présente dans son domaine et ceci *indépendamment* des valeurs prises par les autres variables. Avant d'indiquer les remèdes possibles nous allons d'abord donner un exemple concret d'une telle situation.

4.2.3 Un exemple typique de manque de propagation

Supposons que l'on doive modéliser les contraintes d'ordonnancement suivantes. Soit un ensemble de tâches en disjonction telles que la durée de chaque tâche est fonction de sa date de démarrage. Typiquement, on utilisera une contrainte imposant le fait que les tâches ne peuvent se couper deux à deux. D'autre part, on utilisera également pour chaque tâche une contrainte faisant le lien en son origine et sa durée. Le problème auquel se heurte cette approche vient du fait que la première contrainte, qui impose la disjonction, fait l'hypothèse simplificatrice que la durée d'une tâche est égale à sa durée minimale, et ceci indépendamment de sa date de démarrage. Elle n'a en effet pas accès à cette information qui est encapsulée dans une autre contrainte.

4.2.4 Les remèdes possibles

Ayant bien souvent reçu des plaintes d'utilisateurs de système de programmation par contraintes concernant un manque flagrant de déduction dans une situation particulière une première solution pour remédier à cet état de fait est la suivante. On rajoute des arguments à une contrainte déjà existante ou l'on crée une nouvelle contrainte. Dans le cadre de l'exemple donné précédemment, une solution ad hoc serait l'introduction au cœur de la contrainte disjonctive du lien entre l'origine et la durée de chaque tâche et la modification de l'algorithme de filtrage afin de tirer partie de cette information supplémentaire. Si cette approche permet bien dans la pratique de répondre aux manques ponctuels flagrants de propagation détectés, elle conduit malheureusement à des contraintes « baroques » ayant un nombre élevé d'arguments ad hoc. Un exemple particulièrement extrême de cette approche concerne la contrainte *cycle* avec ses 17 arguments [BOURREAU 99]. De manière paradoxale on revient un peu au problème que l'on voulait éviter initialement, c'est-à-dire de créer un algorithme particulier pour chaque problème que l'on traite.

C'est cette raison qui nous a particulièrement motivés pour rechercher des mécanismes permettant, lorsque l'on écrit un algorithme de filtrage, de mieux tenir compte des contraintes extérieures sans pour autant les connaître explicitement. Pour ce faire nous nous sommes appuyés sur la constatation suivante : la plupart du temps lorsque l'on écrit un algorithme de filtrage associé à une contrainte on est amené à faire implicitement des hypothèses simplificatrices sur les valeurs pouvant être prises par les variables. Si l'on reprend le cas de la contrainte disjonctive imposant le fait qu'un ensemble de tâches ne se coupent pas deux à deux on est amené à élaguer l'origine d'une tâche de manière à ce qu'elle ne coupe pas un intervalle donné. Pour ce faire on fait implicitement l'hypothèse que la tâche en question peut effectivement prendre sa durée minimale. En réalité si l'on voulait être vraiment plus précis, il faudrait d'abord s'interroger sur la durée minimale que peut prendre la tâche sous l'hypothèse qu'elle recoupe l'intervalle dont on veut empêcher le recoupement. Ce constat nous a conduit à proposer une nouvelle primitive prenant la forme «`ask_what_if(Information, Expression, Restriction)`». Cette primitive permet de demander la valeur minimale (`Information=min`) ou maximale (`Information=max`) d'une expression soumise à une liste de restrictions. Chacune des restrictions correspond à une condition portant sur une seule variable. Maintenant, au lieu de directement demander la valeur minimale ou maximale d'une variable en consultant son domaine courant, un algorithme pourra demander ces informations en spécifiant les hypothèses supplémentaires qu'il est localement amené à faire. C'est ainsi qu'il pourra faire des raisonnements plus fins prenant partiellement en compte les autres contraintes.

Dans [BELDICEANU 00b] nous avons montré comment utiliser cette primitive pour revisiter certains algorithmes associés à la contrainte *cumulative*, à la contrainte *cycle* et à certaines contraintes arithmétiques. La mise en œuvre effective d'une telle primitive de communication transparente entre contraintes demanderait la modification du noyau gérant les contraintes. Nous pensons que c'est encore prématuré et qu'il est nécessaire d'étudier d'autres cas où l'on constate un manque flagrant de propagation afin de proposer des mécanismes de communication suffisamment généraux.

4.2.5 Synthèse et conclusion

Nous avons montré qu'un point particulièrement faible de la programmation par contraintes réside dans le fait d'un manque de communication entre les contraintes. De manière plus générale il serait temps de ne plus considérer le fait que l'on écrive un algorithme monolithique pour résoudre un problème particulier sans se soucier du contexte extérieur à cet algorithme. A cet effet, il faudrait développer un style d'écriture d'algorithme dans lequel on rendrait explicite les hypothèses locales que l'on fait, ceci afin de tenir partiellement compte des autres contraintes. Cette approche consiste à rendre un algorithme plus « déclaratif » en rendant explicites des faits qui jusqu'à maintenant ne sont présents qu'en filigrane de l'algorithme. Elle relève finalement de la même perspective que l'idée d'ajouter des explications [JUSSIEN 01] dans les algorithmes de filtrage.

4.3 Mise au point de programmes contenant des contraintes

Notre premier travail concernant la mise au point de programmes contenant des contraintes [SIMONIS, AGGOUN, BELDICEANU & BOURREAU 00] provient de notre participation au projet Esprit Discipl sur ce thème. Par la suite nous avons continué à travailler sur ce thème en généralisant certains aspects de notre travail précédent [ÅGREN, SZEREDI, BELDICEANU & CARLSSON 02] dans le cadre d'un outil de mise au point de contraintes dans SICStus Prolog.

La mise au point de programmes contenant des contraintes n'est pas une tâche aisée pour le développeur d'applications basées sur les contraintes. Cette difficulté a plusieurs sources :

- Du fait de leur exécution comme des coroutines on ne maîtrise pas l'ordre d'exécution des algorithmes de filtrage et le retour arrière (lors de la recherche arborescente) vient encore compliquer cette situation.
- Le développeur d'une application utilisant les contraintes ne maîtrise pas forcément les principes des algorithmes mis en œuvre par les contraintes qu'il utilise.

- Il n'est guère facile d'extraire les événements pertinents de la masse d'informations produites lors de l'exécution des contraintes.
- Il y a bien souvent un écart conceptuel significatif entre les événements générés au niveau du noyau de contraintes (i.e. modification du domaine d'une variable, réveil d'une contrainte) et la manière dont ces événements s'interprètent d'un point de vue applicatif.

Dans le projet Discipl notre démarche pour essayer de répondre à ces problèmes a été la suivante :

- Dans un premier temps nous avons commencé par définir une trace comportant des informations sur la modification des domaines des variables, le réveil des contraintes et l'activation des algorithmes de filtrage. Suite à cela nous avons modifié le noyau de contraintes de manière à ce qu'il produise cette trace.
- Dans un deuxième temps, de la même manière que l'outil de visualisation de OZ, nous avons défini un outil graphique permettant de visualiser l'arbre de recherche et de suivre de manière visuelle les modifications des domaines et le réveil des contraintes aux différents nœuds d'un arbre de recherche.
- Enfin, dans un troisième temps nous avons défini pour chaque contrainte globale des outils de visualisation adaptés aux interprétations courantes attachées à ces contraintes. Par exemple, dans le cadre de la contrainte *cumulative*, au lieu de suivre les modifications des domaines des origines des tâches on visualise l'évolution du profil de consommation de la ressource dans les différents nœuds de l'arbre de recherche. De même, dans le cadre de la contrainte *diffn*, on montre les parties obligatoires des objets avant que ces derniers ne soient complètement placés. Ces programmes de visualisation des contraintes globales sont décrits dans [SIMONIS, AGGOUN, BELDICEANU & BOURREAU 00].

Dans le cadre de la mise au point, les contraintes globales permettent de se rapprocher du niveau applicatif en montrant des vues de haut niveau spécifiques à chaque contrainte globale. Dans cette perpective, la classification des contraintes globales que nous avons faite devrait également s'avérer extrêmement utile : au lieu de redévelopper un algorithme de visualisation spécifique à chaque contrainte globale on imagine facilement que l'on pourrait développer des algorithmes de visualisation génériques basés sur la description explicite des contraintes globales.

4.4 Conclusion

Dans ce chapitre nous venons de présenter nos contributions concernant les aspects déclaratifs des contraintes globales. Nous avons introduit une classification des contraintes en terme de caractéristiques de graphe qui a permis de représenter explicitement le sens de plus d'une centaine de contraintes. Nous avons également abordé le problème de la communication entre contraintes ainsi que le problème de la mise au point. L'étude des aspects déclaratifs est à l'heure actuelle un sujet largement ignoré[42] qui est pourtant fondamental. D'abord il permet un rééquilibrage nécessaire de la programmation par contraintes vers l'intelligence artificielle ; en effet en ignorant complètement cet aspect on retombe dans les travers de l'élaboration de solutions ad hoc pour des problèmes particuliers. Enfin il devrait permettre d'organiser les résultats épars et parcellaires dans une théorie générale faisant le lien entre la forme (i.e. la description explicite des contraintes en terme de constituants élémentaires) et le fond (i.e. les algorithmes de filtrage associés au contraintes).

[42] Mise à part l'élaboration d'outils de mise au point et la génération d'explications.

5 Conclusion

Au terme de ce mémoire marquons une pause pour regarder le chemin parcouru depuis la fin de mes études. Depuis les quinze dernières années la programmation par contraintes a évolué d'une idée de base vers un paradigme utilisé de manière croissante pour résoudre des problèmes combinatoires dans l'industrie. Cette réussite repose sur un mariage réussi et un juste équilibre entre les domaines de l'intelligence artificielle, de la recherche opérationnelle et de l'algorithmique.

A travers les différentes étapes de mon parcours j'ai travaillé dans des environnements qui m'ont indiscutablement permi de m'épanouir dans le domaine des contraintes :

- Dans un premier temps j'ai bénéficié des cours de Jean-Louis Laurière et de Jacques Pitrat dans le domaine des contraintes. Il y avait à cette époque une foi dans l'intelligence artificielle et le séminaire de recherche de Jacques Pitrat était bien souvent une source de réflexion et d'inspiration.
- Par la suite, lorsque j'ai travaillé en Allemagne à l'European Computer Research Centre, il y régnait une atmosphère pleine d'enthousiasme et une émulation autour de la programmation par contraintes.
- Chez COSYTEC j'ai eu la chance d'être en contact avec une grande variété de problèmes qui m'a conduit à m'intéresser à différents domaines. J'ai également appris à considérer le travail des ingénieurs, qui bien que ne disposant pas toujours du temps de réflexion nécessaire comme dans le monde académique, doivent apporter dans des délais rapides des solutions concrètes à des problèmes complexes.
- Enfin à SICS, j'ai pu prendre le temps de la réflexion par rapport à mon travail dans l'industrie et débuter une collaboration fructueuse avec plusieurs universités et avec l'industrie suédoise.

5.1 Résumé de nos contributions

Bien que les contraintes globales remontent à ALICE [LAURIÈRE 76] (contrainte *alldifferent*) et aux débuts de CHIP [VAN HENTENRYCK & CARILLON 88] (contrainte *element*), nous avons été le premier [BELDICEANU 90] à mettre explicitement en avant ce concept de manière consciente et à travailler de manière systématique dessus. En effet les contraintes *alldifferent* et *element* ont été initialement introduites sans mettre l'accent sur la notion de contrainte globale. Comme nous l'avons vu dans le deuxième chapitre les contraintes globales sont, à ce jour, utilisées dans la

majorité des systèmes de programmation par contraintes sur les domaines finis (une dizaine de systèmes à l'heure actuelle, dont huit systèmes industriels).

Dans notre travail sur les contraintes globales nous distinguons les trois phases[43] suivantes :

- La phase «*contraintes globales monolithiques*» pour des familles de problématiques données,
- La phase «*recherche de techniques génériques*» permettant de proposer des algorithmes de filtrage valables pour toute une famille de contraintes,
- La phase «*description explicite*» cherchant à décrire le sens des contraintes globales de manière explicite afin d'ouvrir la voie à la synthèse ou la composition d'algorithmes de filtrage.

La phase «*contraintes globales monolithiques*» concerne principalement l'introduction des contraintes *cumulative* [AGGOUN & BELDICEANU 92], *cumulative-trapèze* [PODER 02], [PODER, BELDICEANU & SANLAVILLE 02], *diffn* et *cycle* dans CHIP et l'introduction de la contrainte *cumulatives* dans SICStus. Bien que ces contraintes aient été utilisées avec succès pour résoudre un nombre important de problèmes industriels, leur algorithmes de filtrage sont ad hoc. En effet ils ne s'appliquent qu'à une contrainte bien précise.

La phase «*recherche de techniques génériques*» concerne la recherche de méthodes générales pouvant, soit s'appliquer à une famille de contraintes, soit servir à l'établissement d'algorithmes de filtrage pour plusieurs contraintes. On peut classer dans cette phase nos travaux plus récents concernant l'opérateur de cardinalité [BELDICEANU & CARLSSON 01a], l'utilisation des méthodes de balayages [BELDICEANU & CARLSSON 01b], [BELDICEANU & CARLSSON 01c], [BELDICEANU, CARLSSON & THIEL 02], [BELDICEANU, CARLSSON & THIEL 03] et l'emploi d'automates [CARLSSON & BELDICEANU 02a], [CARLSSON & BELDICEANU 02b].

Finalement la phase «*description explicite*» est représentée par nos efforts de classification des contraintes globales et de description explicite de leur sens [BELDICEANU 00b], [BELDICEANU 00c]. Cette description, qui s'appuie sur un nombre limité de notions simples du domaine des graphes, nous a permis de décrire le sens de la quasi-totalité des contraintes globales présentes à ce jour. Elle a également mené à la découverte de nouvelles contraintes globales n'existant pas encore à ce jour telles que *change_pair*, *common*, *disjoint* ou *used_by*.

[43] Ces trois phases ne sont bien sûr pas complètement disjointes dans le temps.

5.2 Perspectives de nos travaux

Nous considérons successivement les aspects algorithmiques et les aspects déclaratifs. Puis nous terminons par une synthèse des deux aspects précédents.

5.2.1 Aspects algorithmiques

Pour aboutir à des algorithmes de filtrage efficaces, il est indispensable de travailler très à fond les aspects algorithmiques sous jacents aux trois phases que nous avons mentionnées précédemment, à savoir les «*contraintes globales monolithiques*», la «*recherche de techniques génériques*», et enfin la «*description explicite*».

Contraintes globales monolithiques. Ce paragraphe montre comment il souvent possible de transformer une problématique relevant de l'algorithmique pure en une contrainte globale ad hoc. L'idée centrale étant de casser la distinction entre paramètres en entrée et paramètre en sortie. En effet, il est le plus souvent possible d'associer à un problème P résolu couramment par un algorithme A une contrainte globale C. Pour ce faire on procède comme suit :

– dans le problème P résolu couramment par l'algorithme A on a, d'une part des données en entrée E qui sont complètement fixées lorsque l'on active A, et d'autre part des résultats en sortie S calculés par A à partir des données en entrée E.
– maintenant, dans une perspective contrainte, on replace le problème P par une contrainte globale C(E,S) dans laquelle les données E ne sont a priori plus complètement fixées. De manière similaire les sorties S peuvent être partiellement connues. La contrainte C(E,S) est vérifiée[44] lorsque S correspond au résultat que retournerait A sur les données en entrée E.

La note[45] illustre cette idée avec le problème du tri par ordre croissant d'une suite d'entiers. Bien entendu, des exemples similaires existent également dans les domaines de la géométrie algorithmique et de l'algorithmique sur les mots.

---

[44] Comme il est d'usage en programmation par contraintes on définit une contrainte en se référant au fait que tout ses arguments soient fixés.

[45] Afin de concrétiser cette idée, considérons le problème P consistant à trier par ordre croissant une suite d'entiers E. Un algorithme de tri A prend E en entrée et produit la suite correspondante triée S. Au problème P on peut maintenant associer la contrainte tri(E,S) dans laquelle E et S sont des collections de variables domaine. Cette contrainte impose que S soit triée par ordre croissant et qu'il existe une permutation permettant de passer de E à S. Ainsi l'appel à `tri([2..3,1..2,2..3],[1..2,1..3,1..2])` retournera la solution partielle suivante `tri([2,1..2,2],[1..2,2,2])` correspondant aux deux uniques solutions `tri([2,1,2],[1,2,2])` et `tri([2,2,2],[2,2,2])` associées à l'appel précédent.

Recherche de techniques génériques. La recherche de techniques génériques que nous avons entreprise adaptant des techniques provenant de la géométrie algorithmique ou des automates devrait être poursuivie. Nous considérons également les deux domaines suivants comme particulièrement prometteurs :

– la recherche d'algorithmes de filtrage génériques au sens où les mêmes algorithmes peuvent s'utiliser sur plusieurs types de variables : les variables domaine, les variables ensemble ou les variables «multiset»,
– la conception d'algorithmes de filtrage prenant en paramètres différents type d'algorithmes élémentaires (de la même façon que l'on rend générique un algorithme de tri en lui passant en paramètre une fonction comparant deux items).

Description explicite. La conception d'algorithmes de filtrage pour les contraintes dont le sens peut être décrit explicitement repose en partie sur l'établissement d'algorithmes de graphes prenant en compte les différentes caractéristiques que nous considérons.

Conclusion. De manière générale, on peut cependant regretter que la communauté algorithmique n'accorde pas une importance suffisante au développement d'algorithmes de filtrages. En effet le développement de tels algorithmes est une tâche compliquée qui aurait tout à gagner d'une telle collaboration.

5.2.2 Aspects déclaratifs

Tournons nous vers les aspects déclaratifs liés aux contraintes globales. Ces contraintes peuvent être perçues comme les mots d'un vocabulaire permettant de modéliser des conditions revenant de façon fréquente dans différents problèmes et ceci indépendamment de la technique de résolution mise en œuvre par la suite.

Un des principes fondateurs qui se retrouve dans un grand nombre de mes recherches est le fait que l'on ait toujours à gagner à rendre explicites des informations qui jusqu'à présent étaient enfouies au cœur des algorithmes et des contraintes. C'est ce qui permet d'obtenir des contraintes plus générales pouvant s'utiliser dans un nombre plus varié de contextes. Cela commence par le remplacement du langage naturel pour la description des contraintes par une description explicite de leur sens[46] qui peut être lue par un programme. Cela continue par le fait de rendre explicites des hypothèses locales aux algorithmes de filtrage afin de permettre une meilleure communication avec

[46] Il semble a priori vain d'espérer de pouvoir capturer absolument toutes les contraintes potentielles avec un jeu limité de concepts. Malgré tout nous avons cependant réussi à représenter le sens de la majorité des contraintes globales répertoriées à ce jour.

les contraintes extérieures. Cela enfin se termine par la nécessité de générer des explications sur le pourquoi du retrait des valeurs du domaine par les algorithmes de filtrage.

5.2.3 Synthèse

S'il est maintenant acquis que la programmation par contraintes est un sujet interdisciplinaire par essence, on constate cependant dans la pratique que cela est trop souvent compris comme une simple juxtaposition ou une réappropriation de certaines techniques. En ce qui nous concerne, les aspects algorithmiques et déclaratifs constituent les deux faces d'une même réalité entre lesquelles il s'agit de mettre en évidence les liens. On peut s'étonner qu'à l'heure actuelle ces deux aspects soient typiquement dissociés. Par exemple les aspects composants sont principalement traités dans le cadre des extensions des langages objets tandis que la grande majorité[47] de la communauté recherche opérationnelle ou de la communauté algorithmique résout des problèmes particuliers sans se soucier de l'aspect composant. Cela constitue un frein à la diffusion rapide des résultats pointus provenant de ces deux domaines de recherche dans des composants industriels utilisables pour résoudre une large gamme de problèmes. Pour notre part, les contraintes globales sont à la croisée de ces deux domaines : c'est en levant le voile sur les liens entre ces domaines que l'on pourra vraiment arriver à des composants logiciels conjuguant généralité, efficacité et flexibilité. Cela devrait conduire à une nouvelle génération d'outils industriels d'aide à la résolution de problèmes combinatoires.

[47] Il existe malgré tout certaines équipes dans le domaine de l'algorithmique ayant le souci de réaliser des bibliothèques de composants.

Index

Cet index permet de localiser les différentes contraintes ainsi que les systèmes de programmation par contraintes mentionnées dans ce mémoire. Une contrainte en **gras** renvoie à la définition de la contrainte ; en *italique* il s'agit d'une référence à la contrainte. Pour les systèmes de programmation on indique également en **gras** le paragraphe principal où ils sont présentés.

Bibliographie

[ÅGREN, SZEREDI, BELDICEANU & CARLSSON 02]
M. Ågren, T. Szeredi, N. Beldiceanu et M. Carlsson. Tracing and explaining execution of CLP(FD) programs. Workshop on Logic Programming Environments Copenhagen, Denmark, juillet, 2002.

[AGGOUN & BELDICEANU 90]
A. Aggoun et N. Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. Dans S. Bourgault et M. Dincbas, editors, Programmation en Logique, Actes du 8ième Séminaire, 487-509. France Telecom - CNET, mai 1990.

[AGGOUN & BELDICEANU 92]
A. Aggoun et N. Beldiceanu. Extending CHIP to solve complex scheduling and packing problems. Dans Journées Francophones de Programmation Logique, Lille, France, 1992.

[AGGOUN & BELDICEANU 93a]
A. Aggoun et N. Beldiceanu. Overview of the CHIP Compiler System. Dans Benhamou et Colmerauer (ed), Constraint Logic Programming: chapitre 22, 421-435, The MIT Press, 1993.

[AGGOUN & BELDICEANU 93b]
A. Aggoun et N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. Mathl. Comput. Modelling, 17(7):57-73, 1993.

[AHUJA, MAGNANTI & ORLIN 93]
R.K. Ahuja, T.L. Magnanti et J.B. Orlin. Network Flows : Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[ALTHAUS, BOCKMAYR, ELF, KASPER, JÜNGER & MEHLHORN 02]
E Althaus, A. Bockmayr, M. Elf, T. Kasper, M. Jünger et K. Mehlhorn. SCIL - Symbolic Constraints in Integer Linear Programming. 10th European Symposium on Algorithms, ESA'02, Rome, Septembre 2002. Springer, LNCS 2461, 75-87.

[BAPTISTE, LE PAPE & NUIJTEN 01]
P. Baptiste, C. Le Pape et W. Nuijten. Constraint-Based Scheduling Applying Constraint Programming to Scheduling Problems. Kluwer Academic Publishers, Boston, juillet 2001.

[BAPTISTE, LE PAPE & PERIDY 98]
P. Baptiste, C. Le Pape et L. Peridy: Global Constraints for Partial CSPs: A Case-Study of Resource and Due Date Constraints. Principles and Practice of Constraint Programming - CP'98, 4th International Conference, Pisa, Italy, (26-30 octobre, 1998), Proceedings. Lecture Notes in Computer Science, Vol. 1520, Springer, 87-101, 1998.

[BARNIER & BRISSET 00]
N. Barnier et P. Brisset. Allocation de créneaux pour la régulation du trafic aérien. Neuvièmes Journées Francophones de Programmation Logique et Programmation par Contraintes. Marseille, France, 149-164, juin 2000.

[BELDICEANU 87a]
N. Beldiceanu. Un langage de règles de production basé sur des contraintes et des actions. 7ième Conférence Internationale: les systèmes experts et leurs applications, 1139-1158, Avignon, France, mai 1987.

[BELDICEANU 88]
N. Beldiceanu. Langage de Règles et Moteur d'Inférences Basés sur des Contraintes et des Actions. Application aux Réseaux de Petri. Thèse de l'université de Paris 6, 26 janvier 1988.

[BELDICEANU 90]
N. Beldiceanu. An example of introduction of global constraints in CHIP: Application to block theory problems. Rapport technique TR-LP-49, ECRC, Munich, Allemagne, mai 1990.

[BELDICEANU 98]
N. Beldiceanu. Parallel machine scheduling with calendar rules. Dans 6th International Workshop on Project Management and Scheduling. Istanbul, Turkey, 7-9 juillet, 1998.

[BELDICEANU 00a]
N. Beldiceanu. Sweep as a generic pruning technique. Dans TRICS: Techniques foR Implementing Constraint programming, CP2000, Singapore, 2000.

[BELDICEANU 00b]
N. Beldiceanu. Global Constraints as Graph Properties on a Structured Network of Elementary Constraints of the Same Type. Rapport Technique SICS T2000:01, 2000.

[BELDICEANU 00c]
N. Beldiceanu. Global Constraints as Graph Properties on a Structured Network of Elementary Constraints of the Same Type. Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, 18-21 septembre, 2000, Proceedings. Lecture Notes in Computer Science, (1894), 52-66, Springer , 2000.

[BELDICEANU 01b]
N. Beldiceanu. Pruning for the minimum Constraint Family and for the number of distinct values Constraint Family. Principles and Practice of Constraint Programming – CP 2001, 7th International Conference, Chypre, 26 Nov-1Dec, Springer, 2001.

[BELDICEANU, BOURREAU, RIVREAU & SIMONIS 96]
N. Beldiceanu, E. Bourreau, D. Rivreau et H. Simonis. Solving Resource-constrained Project Scheduling Problems with CHIP. Proc. of Fifth International Workshop on Project Management and Scheduling (PMS'96), Poznan, 35-38, 1996.

[BELDICEANU & CARLSSON 01a]
N. Beldiceanu et M. Carlsson. Revisiting the cardinality Operator and Introducing the cardinality-path Constraint Family. Seventeenth International Conference on Logic Programming ICLP'01, novembre 2001.

[BELDICEANU & CARLSSON 01b]
N. Beldiceanu et M. Carlsson. Sweep as a Generic Pruning Technique Applied to the Non Overlapping Rectangles Constraint. Principles and Practice of Constraint Programming – CP 2001, 7th International Conference, Chypre, 26 Nov-1Dec, Springer, 2001.

[BELDICEANU & CARLSSON 01c]
N. Beldiceanu et M. Carlsson. Sweep as a Generic Pruning Technique Applied to Constraint Relaxation. Soft'01: Modelling and Solving Problems with Soft Constraints, 1 Decembre, Chypre, 2001.

[BELDICEANU & CARLSSON 02]
N. Beldiceanu et M. Carlsson. A New Multi-Resource cumulatives Constraint with Negative Heights. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002.

[BELDICEANU, CARLSSON & THIEL 02]
N. Beldiceanu, M. Carlsson, et S. Thiel. Cost-Filtering Algorithms for the two Sides of the Sum of Weights of Distinct Values Constraint. SICS technical report T2002:14, 2002.

[BELDICEANU, CARLSSON & THIEL 03]
N. Beldiceanu, M. Carlsson et S. Thiel. Sweep Synchronization as a Global Propagation Mechanism. SICS technical report T2003:02, 2003, (accepté dans CPAIOR'03).

[BELDICEANU & CONTEJEAN 94]
N. Beldiceanu et E. Contejean. Introducing Global Constraints in CHIP. Mathl. Comput. Modelling 20 (12), 97-123, 1994.

[BELDICEANU, GUO & THIEL 01]
N. Beldiceanu, Q. Guo et S. Thiel. Non-overlapping Constraint between Convex Polytopes. Principles and Practice of Constraint Programming – CP 2001, 7th International Conference, Chypre, 26 Nov-1Dec, Springer, 2001.

[BENHAMOU et al. 96]
F. Benhamou, P. Bouvier, A. Colmerauer, H. Garetta, B. Giletta, J.L.Massat, G.A. Narboni, S. N'Dong, R. Pasero, J.F. Pique, Touravane, M. Van Caneghem et E. Vétillard. Le manuel de Prolog IV. PrologIA, Marseille, juin 1996.

[BERGE 58]
C. Berge. Théorie des Graphes et ses Applications, Dunod, Paris 1958.

[BERGE 70]
C. Berge. Graphes. Dunod, 1970.

[BERGE 87]
C. Berge. Hypergraphes, Combinatoire des ensembles finis. Dunod, 1987.

[BERNARD, MOUNIER, BELDICEANU & HADDAD 88]
J.M. Bernard, J.L. Mounier, N. Beldiceanu et S. Haddad. AMI, an Extensible Petri Net Interactive Workshop 9th European Workshop on Application and Theory of Petri Nets, Venise, Italie, juin 1988.

[BERTHIER 88]
F. Berthier. Using CHIP to Support Decision Making. Actes du Séminaire 1988 – Programmation en Logique, CNET, Tregastel, France, mai 1988.

[BISDORFF, LAURENT & PICHON 95]
R. Bisdorff, S. Laurent, E. Pichon. Knowledge Engineering with CHIP – Application to a Production Scheduling Problem in the Wire-Drawing Industry. Proceedings of the Third Conference onf Practical Applications of Prolog, 53-62, Paris, avril 1995.

[BOCKMAYR & KASPER 98]
A. Bockmayr et T. Kasper. Branch-and-Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. INFORMS J. Computing, 10/3, 287-300, 1998.

[BOIZUMAULT, DELON & PÉRIDY 93]
P. Boizumault, Y. Delon, L. Péridy. Solving a real life exams problem using CHIP. Logic Programming, Proceedings of the 1993 International Symposium, MIT Press, p.661, octobre 1993.

[BORNING 77]
A. Borning. ThingLab - An Object-Oriented System for Building Simulations Using Constraints. Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, août 1977. William Kaufmann, 1977.

[BOURREAU 99]
E. Bourreau. Traitement de contraintes de graphes en programmation par contraintes. Thèse de l'université Paris 13, 30 mars 1999.

[CARLIER & CHRÉTIENNE 88]
J. Carlier et P. Chrétienne. Problèmes d'ordonnancement, modélisation/complexité/algorithme. Collection E.R.I., Masson, 1988.

[CARLIER & PINSON 90]
J. Carlier et E. Pinson: A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. Annals of Operations Research 26:269-287, 1990.

[CARLSSON, OTTOSSON & CARLSON 97]
M. Carlsson, G. Ottosson et B. Carlson. An Open-Ended Finite Domain Constraint Solver. Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

[CARLSSON & BELDICEANU 02a]
M. Carlsson et N. Beldiceanu. Revisiting the Lexicographic Ordering Constraint. Rapport Technique SICS T2002:17, 2002.

[CARLSSON & BELDICEANU 02b]
M. Carlsson et N. Beldiceanu. Arc-Consistency for a Chain of Lexicographic Ordering Constraints. Rapport Technique SICS T2002:18, 2002.

[CASEAU & LABURTHE 96a]
Y. Caseau et F. Laburthe. Introduction to the Claire Programming Language. LIENS Report 96-15, École Normale Supérieure, 1996.

[CASEAU & LABURTHE 96b]
Y. Caseau et F. Laburthe. Cumulative Scheduling with Task Intervals, Proceedings of the Joint International Conference and Symposium on Logic Programming, MIT Press, 1996.

[CASEAU & LABURTHE 97a]
Y. Caseau et F. Laburthe. Solving Various Weighted Matching Problems with Constraints. Principles and Practice of Constraint Programming - CP'97, 3rd International Conference, Schloss Hagenberg, Austria, (29 octobre - 1 novembre, 1997), Proceedings. Lecture Notes in Computer Science, Vol. 1330, Springer, 17-31, 1997.

[CASEAU & LABURTHE 97b]
Y. Caseau et F. Laburthe. Solving Small TSPs with Constraints. Fourteenth International Conference on Logic Programming - ICLP'97, Leuven, Belgium, (8-11 juillet, 1997), Proceedings. MIT Press, 316-330, 1997.

[CASEAU, LABURTHE & SILVERSTEIN 99]
Y. Caseau, F. Laburthe et G. Silverstein. A meta-heuristic factory for vehicle routing problems. Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, (11-14 octobre, 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, 144-158, 1999.

[COLMERAUER 79]
A. Colmerauer. Sur les bases théoriques de Prolog. Groupe Programmation et Langages AFCET, division théorie et technique de l'informatique, n° 9, 1979.

[COLMERAUER 82]
A. Colmerauer. Prolog II Manuel de Référence et Modèle Théorique, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1982.

[COLMERAUER 90]
A. Colmerauer. An introduction to Prolog III. Communication of the ACM 33(7), 69-91, 1990.

[COLMERAUER 96]
A. Colmerauer. Spécification de Prolog IV. Rapport technique, Laboratoire d'informatique de Marseille, 1996.

[COLMERAUER & ROUSSEL 93]
A. Colmerauer et P. Roussel. The Birth of Prolog. The Second ACM-SIGPLAN History of Programming Languages Conference, 37-52, ACM SIGPLAN Notices, Mars 1993.

[COSTA 94]
M.-C. Costa. Persistency in maximum cardinality bipartite matchings. Operation Research Letters 15, 143-149, (1994).

[COSYTEC 02]
CHIP++ Reference Manual, version 5.4.1, 2002. Cosytec SA, France.

[CREEMERS, GIRALT, RIERA, FERRARONS, ROCA & CORBELLA 95]
T. Creemers, L. R. Giralt, J. Riera, C. Ferrarons, J. Roca et X. Corbella. Constraint-based Maintenance Scheduling on an Electric Power-Distribution Network. PAP'95: Proceedings of the Third International Conference on Practical Applications of Prolog, 135-144, 1995.

[CROCHEMORE, HANCART, LECROQ 01]
M. Crochemore, C. Hancart, T. Lecroq. Algorithmique du texte, Vuibert, Paris, 2001.

[DE BERG, VAN KREVELD, OVERMARS, SCHWARZKOPF 97]
M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Computational Geometry – Algorithms and Applications. Springer, 1997.

[DECHTER & PEARL 87]
R. Dechter, J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. Artificial Intelligence 34, 1-38, 1987.

[DINCBAS, VAN HENTENRYCK, SIMONIS, AGGOUN, GRAF & BERTHIER 88a]
M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf et F. Berthier. The constraint logic programming language CHIP, Proc. Int. Conf. On Fifth Generation Computer Systems FGCS-88, Tokyo, 693-702, 1988.

[DINCBAS, SIMONIS & VAN HENTENRYCK 88b]
M. Dincbas, H. Simonis, P. Van Hentenryck. Solving the Card Sequencing Problem in Constraint Logic Programming. European Conference on Artificial Intelligence (ECAI-88), Munich, Allemagne, août 1988.

[DINCBAS, SIMONIS & VAN HENTENRYCK 88c]
M. Dincbas, H. Simonis, P. Van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. Fifth International Conference of Logic Programming, Seattle, WA, août 1988.

[DUIJVESTIJN 78]
A.J.W. Duijvestijn. Simple Perfect Squared Square of Lowest Order. Journal of Combinatorial Theory, B25, 240-243, 1978.

[ECLIPSE]
Manuel d'utilisation du langage ECL^{i}PS^{e}. (http://www-icparc.doc.ic.ac.uk/eclipse/)

[EPSTEIN, FREUDER, WALLACE, MOROZOV & SAMUELS 02]
S.L. Epstein, E.C. Freuder, R. Wallace, A. Morozov et B. Samuels. ACE, The Adaptive Constraint Engine. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002.

[ERSCHLER & LOPEZ 90]
J. Erschler et P. Lopez. Energy-based approach for task scheduling under time and resources constraints. 2nd International Workshop on Project Management and Scheduling, 115-121, Compiègne, France, juin 1990.

[FAHLE 02]
T. Fahle. Cost Based Filtering vs. Upper Bounds for Maximum Clique. Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02), 93-107, Le Croisic, France, 25-27 mars, 2002.

[FOCCACI, LODI & MILANO 99]
F. Focacci, A. Lodi et M. Milano. Cost-Based Domain Filtering. Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, (11-14 octobre 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, 189-203, 1999.

[FRISCH, HNICH, KIZILTAN, MIGUEL & WALSH 02]
A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel et T. Walsh. Global Constraints for Lexicographic Orderings. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Ithaca, NY, USA, (8-13 septembre, 2002), Proceedings. Lecture Notes in Computer Science, Vol. 2470, Springer, 93-108, 2002.

[FRISCH, HNICH, KIZILTAN, MIGUEL & WALSH 03]
A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel et T. Walsh. Multisets Ordering Constraints. Rapport de recherche APES-55-2003, janvier 2003, (http://www.dcs.st-and.ac.uk/~apes/apesreports.html).

[GAEDE, BRODSKY, GÜNTHER, SRIVASTAVA, VIANU & WALLACE 9V. Gaede, A. Brodsky, O. Günther, D. Srivastava, V. Vianu et M. Wallace (Eds). Constraint Databases and Applications. Lecture Notes in Computer Science 1191. Second International Workshop on Constraint Database Systems, CDB'97, Delphi, Grèce, janvier 1997.

[GALLAIRE 87]
H. Gallaire. Boosting Logic Programming. International Conference on Logic Programming ICLP'87, 962-988, 1987.

[GAMBINI 99]
I. Gambini. Quant aux carrés carrelés. Thèse de l'université de la méditerranée Aix-Marseille II, 18 décembre 1999.

[GENDREAU 99]
M. Gendreau. Constraint Programming and Operations Research: Comments from an Operations Researcher. CRT Pub. No 99-41, (10 pages), octobre 1999.

[GOLTZ & MATZKE 99]
H.-J. Goltz et D. Matzke. University timetabling using constraint logic programming. G. Gupta, editor, Practical Aspects of Declarative Languages, vol. 1551 of Lecture Notes in Computer Science, 320-334, Springer, 1999.

[GRAHAM, LAWLER, LENSTRA & RINNOOY KAN 79]
R. L. Graham, E. L. Lawler, J. K. Lenstra et A. H. G. Rinnooy Kan. 'Optimization and approximation in deterministic sequencing and scheduling: a survey', Ann. Discrete Math., 5, 287-326, 1979.

[GUERNALEC & COLMERAUER 97]
N. Bleuzen Guernalec et A. Colmerauer. Narrowing a 2n-Block of Sortings in O(n log n). Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, (29 octobre – 1 novembre, 1997), Proceedings. Lecture Notes in Computer Science, 2-16, Vol. 1330, Springer, 1997.

[HANAK 02]
Implementing Global Constraints as Structured Networks of Elementary Constraints. The Third Conference of PhD Students in Computer Science. Szeged, Hungary, juillet 2002. (http://www.inf.bme.hu/~dhanak/gcgp_paper.pdf)

[HARIDI, VAN ROY, BRAND & SCHULTE 98]
S. Haridi, P. Van Roy, P. Brand, et C. Schulte. Programming Languages for Distributed Applications. New Generation Computing, n3 v16, 1998, Omsha, Ltd. and Springer-Verlag.

[HNICH 03]
B. Hnich. Function Variables for Constraint Programming. Thèse de l'université d'Uppsala, janvier 2003.

[HOPCROFT & KARP 73]
J.E. Hopcroft et R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing, 2(4):225–231, 1973.

[HOOKER & YAN 02]
J. N. Hooker et Hong Yan, A relaxation for the cumulative constraint, octobre 2001, Révisée en mai 2002. Disponible à: http://ba.gsia.cmu.edu/jnh/papers.html

[IF]
Manuel d'utilisation de IF/Prolog, SIEMENS. (http://www.ifcomputer.com/IFProlog/Constraints/home_en.html).

[JAFFAR, MICHAYLOW, STUCKEY & YAP 92]
J. Jaffar, S. Michaylov, P.J. Stuckey, et Roland H. C. Yap. The CLP(R) language and system. ACM Transactions on Programming Languages and Systems (TOPLAS), 14(3):339-395, juillet 1992.

[JAFFAR & LASSEZ 87]
J. Jaffar et J. L. Lassez. Constraint logic programming. Symposium on Principles of Programming Languages POPL-87, 111-119, 1987.

[JUSSIEN 01]
N. Jussien. Programmation par contraintes avec explications. 7ièmes Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'01), 147-158, 2001.

[KOWALSKI 79]
R. Kowalski. Logic for Problem Solving. North-Holland, 1979.

[KUCHCINSKI 98]
Krzysztof Kuchcinski. An Approach to High-Level Synthesis Using Constraint Logic Programming. Proc. of 24th Euromicro Conference, Workshop on Digital System Design, Västerås, Suède, août 1998.

[LABURTHE 00]
F. Laburthe. Choco: implementing a cp kernel. CP'00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS), Singapore, septembre 2000.

[LAHRICHI 82]
A. Lahrichi. Scheduling: the Notions of Hump, Compulsory Parts and their Use in Cumulative Problems. Dans C. R. Acad. Sc. Paris, t. 294, 209-211, 8 février 1982.

[LAHRICHI & GONDRAN 84]
A. Lahrichi et M. Gondran. Théorie des parties obligatoires et découpes à deux dimensions. Rapport de recherche HI/4762-02 de l'EDF (Électricité de France), (23 pages), janvier1984.

[LAURIÈRE 76]
J.-L. Laurière. Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Thèse de Doctorat d'État de l'université Paris 6, mai 1976.

[LE PAPE 94]
C. Le Pape. Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. Intelligent Systems Engineering, 3(2), 55-66, 1994.

[MEHLHORN & NÄHER 99]
K. Mehlhorn et S. Näher. LEDA A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[MEHLHORN & THIEL 00]
K.Mehlhorn, S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming CP'2000, Lecture Notes in Computer Science, Vol. 1894, 306-319, Springer, (2000).

[MICHEL & VAN HENTENRYCK 02]
L. Michel et P. Van Hentenryck. A Constraint-Based Architure for Local Search. Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, WA, novembre, 2002.

[MILANOE & HOEVE 02]
M. Milano et W.J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002.

[GONDRAN & MINOUX 85]
M. Gondran et M. Minoux. Graphes et Algorithmes (2ième édition revue et augmentée). Eyrolles, Paris, 1985.

[NAREYEK 01]
A. Nareyek. Using Global Constraints for Local Search. Freuder, E. C., et Wallace, R. J. (eds.), Constraint Programming and Large Scale Discrete Optimization, American Mathematical Society Publications, DIMACS Vol.57, 9-28, 2001.

[NIEHREN, TREINEN & TISON 00]
J. Niehren, R. Treinen, et S. Tison. On Rewrite Constraints and Context Unification. Information Processing Letters. Vol. 74 (1-2), 35-40, avril 2000.

[NUIJTEN & LE PAPE 98]
W. P. M. Nuijten et C. Le Pape. Constraint-Based Job Shop Scheduling with ILOG Scheduler. Journal of Heuristics, 3:271-286, 1998.

[OLDER, SWINKELS & EMDEN 95]
W.J. Older, G.M. Swinkels et M.H. van Emden. Getting to the Real Problem : Experience with BNR Prolog in OR. Proceedings of the Third International Conference on the Practical Application of Prolog, (PAP'95 Paris), 465-478, Alinmead Software Ltd, avril 1995.

[OTTOSSON, THORSTEINSSON & HOOKER 99]
G. Ottosson, E. Thorsteinsson et J. N. Hooker, Mixed global constraints and inference in hybrid IP-CLP solvers. CP99 Post-Conference Workshop on Large-Scale Combinatorial Optimization and Constraints, 57-78, 1999.

[OVERMARS & YAP 91]
M. H. Overmars et C.-K. Yap. New upper bounds in Klee's measure problem. SIAM Journal of Computing, 20(6), 1034-1045, décembre 1991.

[PARKER 95]
R. Gary Parker. Deterministic Scheduling Theory. Chapman &. Hall, London, 1995.

[PESANT 01]
G. Pesant. A Filtering Algorithm for the Stretch Constraint. Principles and Practice of Constraint Programming - CP'2001, 7th International Conference, Paphos, Cyprus, (26 novembre - 1 décembre 2001). Lecture Notes in Computer Science, Vol. 2239, Springer, 183-195, 2001.

[PETIT, RÉGIN & BESSIÈRE 01]
T. Petit, J.-C. Régin et C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. Principles and Practice of Constraint Programming - CP'2001, 7th International Conference, Paphos, Cyprus, (26 novembre - 1 décembre 2001). Lecture Notes in Computer Science, Vol. 2239, Springer, 451-463, (2001).

[PETIT, RÉGIN & BESSIÈRE 02]
T. Petit, J.-C. Régin et C. Bessière. Range-Based Algorithm for Max-CSP. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002.

[PION 99]
S. Pion. De la géométrie algorithmique au calcul géométrique. Thèse de l'université de Nice Sophia-Antipolis, 19 novembre 1999.

[PITRAT 90]
J. Pitrat. Métaconnaissance, Futur de l'Intelligence Artificielle. Hermès. 1990.

[PITRAT 93]
J. Pitrat. Penser autrement l'informatique. Edition Hermès, 1993.

[PLATON 03]
PLATON Team. Eclair V8.0, Reference Manual, rapport technique 61 364, 002-108, THALES R & T, Orsay, France, 2003.

[PODER 02]
E. Poder. Programmation par contraintes et ordonnancement de tâches avec consommation variable de ressource. Thèse de l'université Blaise Pascal, 17 octobre 2002.

[PODER, BELDICEANU & SANLAVILLE 02]
E. Poder, N. Beldiceanu et E. Sanlaville. Computing a lower approximation of the compulsory part of a task with varying duration and varying resource consumption. Accepted in European Journal of Operational Research, 2002.

[PREPARATA & SHAMOS 85]
F. P. Preparata et M. I. Shamos. Computational geometry. An introduction. Springer-Verlag, 1985.

[PUGET 92]
J.-F. Puget. PECOS: A high level constraint programming language. Proceedings of the First Singapore International Conference on Intelligent Systems (SPICIS), 137-142, Singapore, Septembre/Octobre 1992.

[PUGET 94]
J.-F. Puget. A C++ implementation of CLP. Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94), Singapore, 1994.

[PUGET 98]
J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. Proceedings of the 15th National Conference on Articial Intelligence (AAAI-98), 359-366, AAAI Press, juillet 1998.

[REFALO 00]
P. Refalo. Linear Formulation of Constraint Programming Models and Hybrid Solvers. Principles and Practice of Constraint Programming - CP'2000, 6th International Conference, Singapore, (18-22 septembre 2000), Proceedings. Lecture Notes in Computer Science, Vol. 1894, Springer, 2000.

[RÉGIN 94]
J.-C. Régin. A filtering algorithm for constraints of difference in CSP. Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 362-367, 1994.

[RÉGIN 96]
J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. Proc. of the Fourteenth National Conference on Artificial Intelligence (AAAI-96), 1996.

[RÉGIN 99a]
J.-C. Régin. The symmetric alldiff constraint. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI’99), Stockholm, Sweden, 420-425, 1999.

[RÉGIN 99b]
J.-C. Régin. Arc Consistency for Global Cardinality Constraints with Costs. Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, (11-14 octobre, 1999), Proceedings. Lecture Notes in Computer Science, Vol. 1713, Springer, 390-404, 1999.

[RÉGIN & RUEHER 99]
J.-C. Régin et M. Rueher. A global constraint combining a sum constraint and binary inequalities. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99), Stockholm, Workshop on Non Binary Constraints, 2 août 1999.

[ROSEAUX 85]
ROSEAUX, Exercices et Problèmes Résolus de Recherche Opérationnelle, T.3, 279-283, Masson, 1985.

[SARASWAT 87]
V. A. Saraswat. Concurrent Logic Programming Languages. PhD thesis, Carnegie-Mellon University, 1987.

[SELLMANN 02]
M. Sellmann: An Arc Consistency Algorithm for the Minimum Weight All Different Constraint. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002.

[SHAW 98]
P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. Principles and Practice of Constraint Programming - CP'98, 4th International Conference, Pisa, Italy, (26-30 octobre, 1998), Proceedings. Lecture Notes in Computer Science, vol. 1520, Springer, 417-431, 1998.

[SIMONIS 90]
H. Simonis. Channel Routing Seen as a Constraint Problem. ECRC Technical Report TR-LP-51, 8, Juin 1990.

[SIMONIS 95]
H. Simonis. The CHIP system and its applications. Principles and Practice of Constraint Programming - CP'95, International Conference, Cassis, France, (19-22 septembre, 1995), Proceedings. Lecture Notes in Computer Science, vol. 976, Springer, 643-646, 1995.

[SIMONIS & BELDICEANU 99]
H. Simonis et N. Beldiceanu A Note on CSPLIB prob007. Problem 007 de la CSPLib, http://csplib.org, 1999.

[SIMONIS, AGGOUN, BELDICEANU & BOURREAU 00]
H. Simonis, A. Aggoun, N. Beldiceanu, E. Bourreau. Complex Constraint Abstraction: Global Constraint Visualisation. Analysis and Visualization Tools for Constraint Programming. Dans Analysis and Visualization Tools for Constraint Programming. Lecture Notes in Computer Science (1870), 299-317, Springer, 2000.

[SIMONIS, BOURREAU & BELDICEANU 99]
H. Simonis, E. Bourreau et N. Beldiceanu. A Note on Perfect Square Placement. Problem 009 de la CSPLib, http://csplib.org, 1999.

[SIMONIS & CORNELISSENS 95]
H. Simonis et T. Cornelissens. Modelling Producer/Consumer Constraints. Principles and Practice of Constraint Programming - CP'95, International Conference, Cassis, France, (19-22 septembre, 1995), Proceedings. Lecture Notes in Computer Science, vol. 976, Springer, 449-463, 1995.

[SIMONIS, NGUYEN & DINCBAS 88]
H. Simonis, H.N. Nguyen et M. Dincbas. Verification of Digital Circuits Using CHIP. Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification, Glasgow, Scotland, juillet 1988.

[SMOLKA 95]
G. Smolka. The Oz Programming Model. J. van Leeuwen, editor, Computer Science Today: Recent Trends and Developments. Springer, LNCS 1000, 324-343, 1995.

[SMOLKA 96]
G. Smolka. Constraints in OZ. ACM Computing Surveys, Vol 28, No 4, décembre 1996.

[SOUISSI & BELDICEANU 88]
Y. Souissi et N. Beldiceanu. Deterministic systems of sequential processes: theory and tools. Dans F.H. Vogt, editor, Concurrency 88, 380-400, LNCS 335, Springer-Verlag, 1988.

[SCHRIJVER 86]
A.Schrijver. Theory of Linear and Integer Programming, Wiley, 1986.

[STEELE & SUSSMAN 80]
G. Steele et G.J. Sussman. CONSTRAINTS – A Language for Expressing Almost Hierarchical Descriptions, Artificial Intelligence 14 (1), 1-39, 1980.

[SUTHERLAND 63]
I.E. Sutherland. SketchPad: A Man-Machine Graphical Communication System. Thèse du Massachusetts Institute of Technology, janvier 1963.

[TUCKER 80]
A. Tucker. Applied Combinatorics. John Wiley & Sons, Inc, 1980.

[VAN HENTENRYCK 89]
P. Van Hentenryck. Constraint Satisfaction in Logic Programming. The MIT Press, 1989.

[VAN HENTENRYCK 99]
P. Van Hentenryck. The OPL Optimization Programming Language. The MIT Press, 1999.

[VAN HENTENRYCK & CARILLON 88]
P. Van Hentenryck et J.-P. Carillon. Generality versus Specificity: an Experience with AI and OR Techniques. American Association for Artificial Intelligence (AAAI-88), St. Paul, Mi, août 1988.

[VAN HENTENRYCK & DEVILLE 91]
P. Van Hentenryck et Y. Deville. The Cardinality Operator: A New Logical Connective for Constraint Logic Programming. ICLP 1991, 745-759, 1991.

[VAN HENTENRYCK, SARASWAT & DEVILLE 95]
P. Van Hentenryck, V. Saraswat, Y. Deville. Design, Implementation and Evaluation of the Constraint Language cc(FD). A. Podelski, ed., Constraints: Basics and Trends, vol. 910 of Lecture Notes in Computer Science, Springer-Verlag, 1995.

[YUNES 02]
T.H.Yunes: On the Sum Constraint: Relaxation and Applications. Principles and Practice of Constraint Programming - CP'2002, 8th International Conference, Cornell University, Ithaca, NY, USA, 8-13 septembre, 2002).

[ZHOU 97]
J. Zhou. A permutation-based approach for solving the job-shop problem. Constraints, 2(2), 185-213, 1997.